Descriptions de scènes multimédia : représentations et optimisations

Cyril Concolato

Thèse

présentée pour obtenir le grade de docteur
de l'Ecole Nationale Supérieure des Télécommunications

Spécialité : **Informatique et Réseaux**

Cyril CONCOLATO

Descriptions de scènes multimédia : représentations et optimisations

Soutenue le 12 juillet 2007 devant le jury composé de :

Yves Matthieu	Président
Fernando Manuel Bernardo Pereira	Rapporteur
Rik Van de Walle	Rapporteur
Françoise Prêteux	Examinateur
Nabil Layaïda	Examinateur
Francesco Morán Burgos	Examinateur
Jean-Claude Dufourd	Directeur de thèse

Remerciements

Je tiens tout d'abord à remercier les différents membres de mon jury pour leur participation et leurs critiques constructives. Je remercie plus spécialement Mme. Françoise Prêteux pour avoir présidé ce jury ; MM. Fernando Manuel Bernardo Pereira et Rik Van de Walle, pour avoir accepté, même au pied levé, d'être les rapporteurs de ma thèse ; et enfin, MM. Nabil Layaïda, Francisco Moran Burgos et Yves Mathieu pour avoir examiné avec soin mon travail.

Je tiens tout particulièrement à remercier mon directeur de thèse, M. Jean-Claude Dufourd, qui m'a permis de faire cette thèse; qui m'a transmis de nombreuses connaissances techniques et non-techniques, qui m'a initié au petit monde du multimédia en France, à celui des comités de standardisation ; et qui, malgré son implication chez Streamezzo, a trouvé le temps pour encadrer cette fin de thèse.

Je voudrais également remercier tous les membres passés et présents de l'équipe qui ont tous contribué à ce travail par les discussions que nous avons eu : Frédéric, Souhila, Gianluca, Mariam, Jean, Jean-Claude, Benoît, Philippe, Frédéric, Clément, Berthele. Je ne peux les citer tous, mais je voudrais également remercier tous les doctorants, stagiaires, enseignants-chercheurs et administratifs des départements COMELEC et TSI que j'ai côtoyés jusqu'à présent à l'ENST, pour l'ambiance amicale et sympathique qu'ils contribuent à créer, en particulier: Béatrice, Christophe, Grégoire et tous ceux que j'oublie. Il me faudrait également remercier tous les collègues des projets européens et nationaux ainsi que des comités de standardisation avec qui j'ai collaborés pendant ces années, mais la liste serait trop longue.

Enfin, je souhaiterais remercier ma famille, mes parents et Laure, qui même s’ils ne comprenaient pas ce que je faisais, m’ont toujours soutenu et encouragé.

Résumé

Avec les progrès technologiques en matière de débits disponibles et la diversification des média disponibles, l'enjeu est maintenant de diffuser aux utilisateurs des médias de qualité dans une présentation cohérente, attractive et interactive, qu'elle soit destinée au monde de la télévision numérique, de l'Internet ou de la téléphonie mobile. Les descriptions de scènes multimédia tentent de répondre à ce pari. Le pari consiste en effet à développer des technologies permettant la diffusion et la présentation de contenu multimédia riche, animé, interactif dans des environnements aux contraintes multiples. Il faut des technologies suffisamment simples pour être intégrées dans des terminaux à faible coût de revient, mais également suffisamment attractives pour attirer de nouveaux consommateurs. Ensuite, même si les débits augmentent et les techniques de compression s'améliorent, la plus grande partie des "tuyaux" est encore occupée par les média traditionnels (audio, vidéo). Il faut donc que ces nouvelles technologies n'occupent pas trop de bande passante. Enfin, pour qu'elles mènent à de nouveaux usages, il faut que les créateurs de contenu se les approprient et pour cela, il faut que les présentations multimédia soient faciles à créer, quel que soit le scénario d'utilisation.

Cette thèse a pour objectif dans un premier temps de mener une réflexion sur les différentes représentations de scènes multimédias existantes et d'en proposer une synthèse. Cette synthèse propose une vue générale sur les langages, les formats de représentation des descriptions de scènes ainsi que sur les techniques de codage et les traitements appliqués aux scènes multimédia. Elle présente, en particulier, les problématiques liées à l'animation, la compression, l'interactivité, la diffusion et finalement à la présentation. Dans un second temps, elle propose des améliorations aux solutions existantes. Ces propositions sont doubles. Tout d'abord, cette thèse propose d'un ensemble de méthodes pour permettre la représentation efficace de scènes multimédia fortement animées, notamment pour les formats de représentations MPEG-4 BIFS, W3C SVG et MPEG-4 LASeR, et avec la prise en compte des contraintes de la diffusion sur téléphone mobile. Ensuite, cette thèse propose une architecture et une implémentation innovante d'un lecteur de scènes permettant la lecture de contenus multimédia sur terminaux contraints grâce au respect d'un compromis entre consommation mémoire et rapidité d'exécution. Cette thèse se conclut, après avoir rappelé les contraintes du monde mobile, par une synthèse des choix technologiques à faire, dans la vaste bibliothèque d'outils disponibles dans le domaine des descriptions de scènes, pour concevoir un service multimédia pour mobiles.

Table des matières

Liste des figures

Liste des exemples de code

Liste des tableaux

Chapitre 0 Introduction

0.1 Exposé des problèmes

L'augmentation des débits sur les réseaux de télécommunication d'une part, l'amélioration des techniques de compression réduisant la bande passante nécessaire pour transmettre de l'information audiovisuelle d'autre part, et enfin la diversification des sources d'informations et des types de média diffusés, notamment sur Internet, ont engendré le besoin de structurer l'information, notamment l'information audiovisuelle. L'enjeu est de diffuser aux utilisateurs des médias de qualité dans une présentation cohérente, attractive et interactive, qu'elle soit destinée au monde de la télévision numérique, de l'Internet ou de la téléphonie mobile. Les descriptions de scènes multimédia tentent de répondre à ce pari.

Le pari consiste en effet à développer des technologies permettant la diffusion et la présentation de contenu multimédia riche, animé, interactif dans des environnements aux contraintes multiples. Il faut des technologies suffisamment simples pour être intégrées dans des terminaux à faible coût de revient, mais également suffisamment attractives pour attirer de nouveaux consommateurs. Ensuite, même si les débits augmentent et les techniques de compression s'améliorent, la plus grande partie des "tuyaux" est encore occupée par les média traditionnels (audio, vidéo). Il faut donc que ces nouvelles technologies n'occupent pas trop de bande passante. Enfin, pour qu'elles mènent à de nouveaux usages, il faut que les créateurs de contenu se les approprient et pour cela, il faut que les présentations multimédia soient faciles à créer, quel que soit le scénario d'utilisation.

0.2 Objectifs

Cette thèse s'intitule "Descriptions de scènes multimédia : représentations et optimisations". Ses objectifs ont été dans un premier temps de mener une réflexion sur les différentes représentations de scènes multimédias et d'en proposer une synthèse : en termes de fonctionnalités présentes ou non [08], de techniques de représentation et enfin de scénarios d'utilisations.

Sur la base de ces réflexions, il a été question dans un second temps de proposer des améliorations aux solutions présentées. Ces améliorations se sont placées au niveau des fonctionnalités des formats de

représentations MPEG-4 BIFS, W3C SVG et MPEG-4 LASeR, se focalisant sur des algorithmes de traitement de ces représentations et des méthodes de codage pour les adapter à divers scénarios d'utilisation avec dernièrement la prise en compte de la diffusion sur téléphone mobile.

0.3 Cadre de travail

Cette thèse a été réalisée dans le groupe « Outils et Services Multimédia » au sein du département COMELEC puis dans le groupe « Multimédia » du département TSI, tous deux à l'ENST Paris. Mes activités de recherche au cours de cette période se sont déroulées parallèlement à de nombreux projets de recherche, auxquels j'ai participé, plus ou moins activement, mais qui m'ont tous donné matière à réflexion. Parmi ces projets, on trouve un projet PRIAMM, qui m'a permis de faire mes premiers pas dans le monde des dessins animés vectoriels. Je citerai également les autres projets nationaux RIAM SAMP4 et MP4MC ; les projets de recherche européens : SoNG, ISIS, DANAE, MELISA et TIRAMISU ; et enfin, des projets industriels, notamment en collaboration avec les équipes de recherche de France Telecom R&D. Enfin, ma thèse ne serait pas ce qu'elle est, si je n'avais pas été impliqué dans la normalisation tout d'abord au sein de MPEG et plus récemment au sein du W3C.

0.4 Principales Contributions

Cette thèse propose une vue générale sur les langages, les formats de descriptions de scènes, sur les techniques liées à ces descriptions et sur les traitements appliqués aux scènes multimédia. Elle présente les problématiques liées à l'animation, la compression, l'interactivité, la diffusion et finalement la présentation, pour, je l'espère, permettre d'appréhender les difficultés existantes dans les formats actuels et ainsi faire un choix éclairé sur le meilleur format dans une situation d'utilisation particulière.

Cette thèse présente également un ensemble de contributions techniques, d'améliorations proposées aux formats existants afin de satisfaire efficacement de nouveaux cas d'utilisation. Ces contributions sont les suivantes :

- la définition et l'ajout au langage BIFS de mécanismes graphiques avancés tels que les gradients de couleurs ;
- un ensemble de règles permettant la structuration des scènes BIFS, SVG ou LASeR et l'utilisation des mécanismes existants de compression afin de réduire efficacement le débit de ces scènes ;
- un outil pour le contrôle de la mémoire nécessaire à la lecture de scènes SVG de longue durée ainsi qu'un mécanisme de fragmentation permettant conjointement la lecture et le streaming de ces scènes ;

- un mécanisme de quantification de graphismes vectoriels alliant le meilleur des technologies BIFS et LASeR ;
- un mécanisme de codage de structure des scènes LASeR offrant de meilleurs résultats que la norme ;
- un mécanisme permettant la scalabilité de scènes graphiques 2D à base de mise à jour ;
- et enfin, une architecture logicielle innovante pour la lecture et la visualisation optimisée de scènes SVG.

0.4.1 Publications et brevets

Plusieurs publications ont été effectuées sur les travaux décrits dans cette thèse. Elles sont rappelées à la fin de ce document. On peut citer notamment deux articles publiés dans les revues *IEEE Transactions on Computer Science and Video Technologies* [03] et *IEEE Multimedia* [02], et cinq articles acceptés à des conférences nationales et internationales. Enfin, cette thèse a abouti également au dépôt de deux brevets internationaux [14][15].

0.4.2 Contributions aux standards

Durant cette thèse, j'ai été amené à participer aux activités de normalisation au sein du groupe MPEG. Mes activités ont principalement porté sur les aspects systèmes des normes MPEG-4, MPEG-7 et MPEG-21. J'ai plus particulièrement participé à la norme MPEG-4 BIFS, notamment en tant qu'initiateur et éditeur de l'amendement *Advanced Text and Graphics*; et plus récemment à la norme MPEG-4 LASeR. Ma participation à MPEG au cours de ces cinq dernières années m'a permis d'écrire ou de participer à plus de quarante contributions. Enfin, ma participation récente au W3C m'a permis de m'impliquer dans les activités du groupe de travail SVG et récemment sur les activités CDF, XBL et REX.

0.4.3 Logiciels

Cette thèse fût également l'occasion de nombreux développements logiciels permettant de mieux comprendre certains aspects des descriptions de scènes, aussi bien au niveau création, encodage, distribution ou lecture. Je citerai la participation aux logiciels ENST d'encodage BIFS « MP4Tool » et de création de contenu BIFS « MPro » [10]. J'ai participé au développement du logiciel de référence de la norme MPEG-4 BIFS notamment en y intégrant une partie des outils que nous proposions pour standardisation.

De nombreux développements ont eu pour but le transcodage entre formats de représentation de scènes. J'ai principalement développé des transcodeurs de contenu au format Flash dans les formats W3C SVG, MPEG-4 XMT, MPEG-4 LASeR XML ; mais également certaines parties de transcodeurs

de contenus SVG en contenus XMT. J'ai également développé un transcodeur de flux de sous-titre au format 3GPP *Timed Text* en BIFS, pour prouver cette faisabilité. Enfin, pour finir sur les transcodages, j'ai initié et participé à un projet visant le transcodage de DVD en MPEG-4, notamment de la partie interactive, dans le format BIFS.

Dans le cadre des divers projets européens, j'ai eu à réaliser un encodeur BIFS scalable, conforme aux normes MPEG-4 et MPEG-21.

Enfin, dans le cadre du projet *Open Source* GPAC, j'ai principalement participé à l'intégration dans le moteur de composition de scènes du format SVG. J'ai également participé aux modules d'encodage/décodage conforme à la norme MPEG-4 LASeR et au module de création/lecture de fichier MPEG-21.

0.5 Plan

Ce document de thèse se décompose en deux parties. La première partie décrit mon analyse de l'état de l'art dans le domaine des descriptions de scènes. Elle se compose de 5 chapitres. Dans le chapitre 1, je présenterai les concepts principaux nécessaires à la compréhension des descriptions de scènes. Je donnerai également un inventaire des principaux formats. Le chapitre 2 présentera plus précisément les aspects liés à l'animation dans les descriptions de scènes. Le chapitre 3 se consacrera aux problématiques de compression. Dans le chapitre 4, j'aborderai les descriptions de scènes du point de vue des mécanismes d'interactivité. Ensuite, dans le chapitre 5, je présenterai les problèmes liés à la création et à la diffusion de contenu multimédia exploitant une description de scène.

La seconde partie de ce document présente les contributions de cette thèse dans le domaine des descriptions de scènes. Elle se décompose en deux chapitres. Le chapitre 6 propose un ensemble de méthodes pour permettre la représentation efficace de scènes multimédia fortement animées. Le chapitre 7 propose une architecture et une implémentation innovante d'un lecteur de scènes permettant la lecture de contenus multimédia sur terminaux contraints grâce au respect d'un compromis entre consommation mémoire et rapidité d'exécution.

Enfin, le chapitre 8 donnera les conclusions de cette thèse sous la forme d'une synthèse et d'une sélection des meilleures technologies de descriptions de scènes pour environnement mobile. Il présent également les perspectives de recherches futures.

Trois annexes complètent ces chapitres : la liste de publications effectuées durant cette thèse, la liste des références bibliographiques citées dans ce document; et, pour les personnes néophytes, un glossaire des trop nombreux acronymes que l'on pourra trouver dans ce document.

Chapitre 1 Principes des descriptions de scènes

1.1 Introduction

Cette thèse traite de différents aspects relatifs aux représentations de scènes multimédia. Il est donc nécessaire en premier lieu de définir un certain nombre de termes, utiles à la compréhension des notions abordées dans le reste de ce document. Ces définitions sont données dans la première partie de ce chapitre. Dans la partie suivante, nous présentons brièvement les principaux langages de descriptions de scènes auxquels nous nous sommes intéressés. Enfin, dans la dernière partie de ce chapitre, nous présentons quelques concepts importants à la compréhension des chapitres suivants.

1.2 Vocabulaire

Nous définissons dans cette partie les termes : scène, description de scène, contenu multimédia, document, flux de scène et arbre de scène. Ces termes seront utilisés dans ce chapitre et dans les chapitres suivants.

1.2.1 Scène

Pour comprendre le sens du mot « scène » utilisé dans cette thèse, nous commençons par une analogie avec le monde du cinéma, dans lequel le mot scène a un sens relativement connu. Un film peut en effet être découpé en scènes. Chaque scène est le résultat de l'évolution de personnages sur un décor selon les choix du metteur en scène.

Dans le monde du multimédia, un contenu multimédia est l'analogue d'un film. Il s'agit par nature d'une agrégation cohérente de contenus (les personnages) de types différents : d'images, de texte, de vidéo … qui sont présentés ou composés ensemble selon les choix définis par l'auteur. Par analogie, dans ce document, nous parlerons de scène audio-visuelle ou scène multimédia pour décrire, d'une part, l'ensemble des éléments qui participent à une présentation multimédia (l'équivalent du *casting* d'un film), et d'autre part, l'agencement de ces éléments pour former une présentation cohérente, interactive et dynamique (l'équivalent de la mise en scène).

Si l'on s'intéresse maintenant au domaine du codage vidéo, quand il s'agit de coder une scène vidéo, certains travaux se sont essayés à coder la scène sous forme d'objets indépendants [16]. Cela permet de coder avec plus de détail les personnages que le fond, qui lui change moins souvent. Cependant, comme les dispositifs de capture (caméra) ne font pas la distinction entre le fond et les personnages, le codage par objet vidéo suppose une phase de segmentation. Cette étape est souvent difficile à réaliser, sauf dans les cas de capture sur fond uniforme (*ChromaKeying*).

Dans le domaine du codage de contenu multimédia, certains objets sont plus importants que d'autres et nécessitent plus de précision également. Cependant, les objets qui composent la scène étant de types différents, on peut naturellement les coder séparément. Une particularité du codage de contenu multimédia est que les instructions de mise en scène des objets de la scène sont elle-même codées séparément des objets audio visuels qu'elles utilisent. Un contenu multimédia sera donc défini comme l'agrégation de medias audiovisuels et d'instructions de mise en scène.

1.2.2 Description de scène

De manière intuitive, un langage de description de scène est un langage qui permet de décrire les instructions de mise en scène, c'est-à-dire comment la scène est composée et comment elle évolue. Plus précisément, nous adoptons la même vision que S. Boll [17], nous considérerons un langage de description de scène comme un langage fondé sur quatre piliers :

- l'utilisation d'objets[1] ou médias audio-visuels :
    - soit numérisés à partir de sources analogiques, enregistrements vidéo issus de la capture par une caméra, images fixes prises par un appareil photo ou morceau musical ou parole issus d'un enregistreur de son,
    - soit dits "synthétiques", c'est-à-dire créés par des procédés informatiques. Parmi les exemples d'objets synthétiques, on peut citer les objets graphiques en deux ou trois dimensions, la parole générée à partir de texte (*text to speech*) ou les musiques générées de type MIDI ou SAOL [18];
- l'organisation spatiale de ces différents objets : la taille de la fenêtre de visualisation de la scène, la définition d'un repère en deux ou trois dimensions pour positionner les objets, et enfin, la position d'un objet par rapport à un autre ou dans ce repère;

[1] Le terme objet utilisé ici ne fait pas référence à la notion d'objet définie dans la norme *MPEG-4 Systems*.

- l'organisation temporelle de ces objets, c'est-à-dire quand un objet est animé, comment il évolue; ou lorsqu'il s'agit d'un média continu, à quel moment il démarre ou se termine; et éventuellement, les contraintes de synchronisation entre ces différents éléments;
- les possibilités d'interactivité que l'utilisateur peut exploiter pour faire évoluer la scène : les types de périphériques d'interaction, les types d'évènements utilisables.

1.2.3 Flux et document de scène

Dans le contexte de la distribution de contenu multimédia, nous nous intéressons ici à la méthode utilisée pour décrire et transmettre la description de scène par un serveur pour un ou plusieurs clients. Parmi les méthodes de descriptions de scènes, on distingue les langages où la description de la scène se présente sous forme d'un document et ceux où la description de la scène est modélisée sous forme de flux.

La représentation sous forme de document concerne en général les scènes statiques, ou les scènes animées où tous les états de l'animation sous connus à l'avance par le lecteur de contenu. Le chapitre 2 décrira plus en détail cette notion d'animation. L'origine du terme 'document' provient du fait que pour ces types de scènes, l'information apportée par la scène est structurée en une seule entité, un seul bloc. Ce bloc généralement est stocké et/ou transmis sous la forme d'un fichier XML [54]. Or, dans la terminologie XML, l'information structurée contenue dans un fichier XML est décrite par la notion de document [60]. Ainsi, quand un fichier XML décrit une scène multimédia, on parle de document représentant une scène multimédia.

La représentation sous forme de flux s'applique, elle, à certaines scènes animées dont l'animation est décrite par morceaux, par trames. Nous verrons également dans le chapitre 2 différentes manières de décrire ces animations. Quand la scène est décrite par un flux de description de scène, terme issu de la norme *MPEG-4 Systems* [28][38][42], on considère une description de scène comme une suite d'unités de base formant un flux, à l'image des flux vidéo ou audio. La Figure 1.1 illustre cette notion de flux de description de scène.

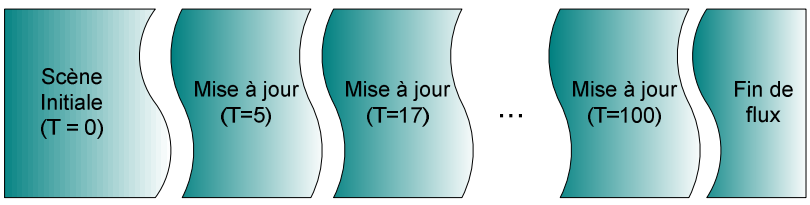


Figure 1.1 – Composition d'un flux de description de scène

L'intérêt de ce modèle sous forme de flux est la possibilité d'intégrer ce flux dans les mécanismes de diffusion de flux temporels classiques, c'est-à-dire avec les mêmes mécanismes que pour les flux vidéo ou audio, ce qui donne une plus grande souplesse pour distribuer le contenu que l'approche utilisant un

document. On peut notamment utiliser les formats de fichier multimédia (MP4 [45], 3GP [73], AVI), les protocoles de *streaming* (SDP [74], RTP [75]) ou les protocoles de diffusion comme MPEG-2 TS [40]. Nous aborderons ces sujets plus en détail dans le chapitre 6.

Il existe une relation complexe entre les notions de flux de scène et document de scène. On peut utiliser un flux de scène pour décrire une scène statique, qui serait représentée naturellement par un document. Pour cela, il suffit de considérer le document comme la seule et unique unité du flux. On peut également représenter un flux sous la forme d'un document XML, où chaque mise à jour est décrite par un élément XML et où le document résultant est la concaténation des éléments décrivant la scène initiale et des éléments décrivant les mises à jour.

En résumé, il faut donc retenir les points suivants que nous exploiterons tout au long de cette thèse :

- un flux de scène peut décrire tous les types de scènes : statiques ou animées. Il est particulièrement adapté pour la distribution;
- un document XML peut décrire toutes les formes de scènes : document ou flux, avec dans ce dernier cas une particularité relative à la notion d'arbre de scène. Cette particularité est décrite dans la partie suivante.

1.2.4 Arbre de scène

Selon la norme W3C DOM [60][84], on peut représenter un document XML par un arbre, dit arbre DOM, dont les nœuds sont principalement les éléments du document XML, le contenu textuel des éléments, les commentaires ou les attributs des éléments XML. Ainsi quand un document XML décrit une scène multimédia, l'arbre DOM est une représentation possible de cette scène.

Il existe de nombreuses représentations arborescentes pour représenter une scène. Le standard DOM en est une. Le standard SVG 1.2 Tiny [50] définit une autre représentation arborescente possible, nommée MicroDOM [55]. Le langage VRML en définit également une troisième. En revanche, dans le langage de publication SWF associé à l'outil auteur Flash [85], la notion d'arbre de scène n'est pas directement apparente. La représentation de la scène dans ce langage est une représentation très simple utilisant deux concepts : le dictionnaire et la liste d'affichage. Le dictionnaire est un ensemble de symboles graphiques (textes, objets vectoriels, tableau de pixels). La liste d'affichage est une liste des symboles visibles à l'écran auxquels sont associés une profondeur, une matrice de transformation de position, une matrice de transformation de couleur et un rectangle de visualisation (*clipping*). Cependant, on peut voir cette représentation comme une restriction de la notion d'arbre de scène. On peut considérer un arbre à deux branches : la première correspond au dictionnaire, non visible à l'écran, et la seconde correspond à la liste d'affichage.

Quand une scène est représentée par un flux, le flux décrit les états de la scène à des instants différents. Dans ce cas, l'arbre de scène évolue avec les mises à jour de la scène contenues dans le flux. Un flux de scène ne décrit donc pas un seul arbre de scène, mais les états successifs de cet arbre lors de la lecture ou de la diffusion.

Il faut remarquer que dans le cas d'une scène, représentée sous la forme d'un flux, décrit par un document XML, l'arbre DOM correspondant à ce document n'est pas l'arbre de scène. Cela peut être problématique dans certains cas, notamment concernant la gestion des identifiants. Considérons l'exemple suivant. A un instant T dans le flux, un nœud de l'arbre, d'identifiant N1, est supprimé. A un autre instant T+dt, un nouveau nœud, d'identifiant N1, est introduit dans l'arbre de scène. Dans ce cas, l'arbre de scène contient, à deux instants différents, des nœuds différents de même identifiant. Cependant, l'arbre DOM décrivant ce flux possèdera deux éléments avec le même identifiant, violant ainsi l'unicité des identifiants DOM.

En résumé, de manière générale, dans le cas de scène sous forme de document ou sous forme de flux, on parlera d'arbre de scène pour décrire l'état à un instant donné de la scène, indépendamment du langage de description utilisé.

1.3 Usages des descriptions de scènes

Nous venons de définir un certain nombre de termes liés aux descriptions de scènes. Grâce à ces définitions, nous pouvons aborder l'état de l'art des techniques de représentation des descriptions de scènes. Cependant, il nous paraît important, avant cela, de donner un aperçu des cas d'utilisation des langages de descriptions de scènes.

Les utilisations possibles d'une description de scènes sont multiples : portail multimédia, guide de programmes pour télévision interactive, jeux vidéo interactifs, *slideshow* musical, application *e-learning*, *video jockeying*, etc. On peut cependant distinguer deux utilisations différentes selon le nombre et l'importance des média audio-visuels.

Dans le cas où la scène est composée autour d'un élément audio-visuel prépondérant, l'utilisation d'une description de scène sert généralement à apporter une surcouche par rapport à cet élément audio-visuel. Cette surcouche peut être graphique (affichage de sous titre, d'informations sur le programme en cours), interactive (navigation dans les chapitres d'un élément audio-visuel comme dans un DVD) et animée (poursuite d'objets).

Dans cette configuration, la description de scène apporte une information dont on peut se passer si on souhaite uniquement présenter le média audio-visuel. Dans un contexte de scalabilité [41], on parlerait de la couche de base de la description de scène, qui ne contiendrait que les directives pour jouer le contenu audio vidéo, et de la couche d'amélioration de la scène qui contiendrait la partie interactive,

graphique, animée … Une telle architecture de scène est notamment adoptée par la norme T-DMB [80] dans le cadre de la télévision numérique interactive terrestre en Corée .

A l'opposé, dans les cas où une scène comporte de nombreux objets audio-visuels et une grande interactivité, on ne peut alors se passer de la description de scène pour présenter le contenu. L'architecture de la scène est ainsi beaucoup plus complexe qu'une architecture en deux couches.

Le portail de vidéo à la demande de la Figure 1.2, réalisé par un étudiant de l'université belge de Gand [19], est un exemple illustrant ce cas. Douze vidéos sont disponibles à la visualisation dans deux qualités différentes. Différentes pages permettent de sélectionner le contenu à visualiser. La description de scène, dans ce contenu, est indispensable pour permettre le positionnement des objets image, la description de l'interactivité (sélection d'une bande annonce en fonction du clic sur une image), de la navigation (choix de la qualité vidéo), etc.

Figure 1.2 – Exemple de portail vidéo à la demande

On voit ici deux types d'usage bien différents des descriptions de scène. Ces usages sont à notre avis intimement liés à l'environnement de consommation du contenu. Dans le cas d'un service Web, où l'utilisateur est habitué à beaucoup d'interactivité, où le contenu est consommé sur un PC à la puissance quasi illimitée, un auteur préférera un scénario du deuxième type, alors que dans le cas d'un service de télévision interactive, où le média principal est le flux de télévision (audio, vidéo), où l'utilisateur ne souhaite pas nécessairement d'interactivité (consommation passive), un auteur préférera le premier scénario.

1.4 Etat de l'art des langages existants

La plupart des langages de description de scène existants permettent de réaliser des scènes décrivant les usages précédents. Nous présentons ici brièvement parmi ces langages, ceux que nous avons étudiés, manipulés et qui nous ont servi dans nos travaux de recherche. Leurs caractéristiques générales sont décrites ici et les caractéristiques précises relatives à la compression, l'animation, l'interactivité seront décrites dans les chapitres suivants. Nous ne présentons ici que les langages déclaratifs. Le lecteur intéressé par les langages de programmation de scènes pourra se référer, entre autre, aux architectures MHP [21][89] ou OpenTV [20][88].

1.4.1 HTML et DHTML

Le langage HTML, *HyperText Markup Language*, [48], développé dans les années 1990, est sans doute le premier langage déclaratif qui peut être assimilé à un langage de description de scène. En effet, HTML a été conçu comme un langage de balises permettant de décrire des pages, au départ comportant uniquement du texte, des images, des liens de navigation … puis, les versions successives du langage ont apporté la possibilité d'inclure dans les pages : des animations (par l'intermédiaire de l'utilisation du langage de script *ECMAScript* [70], on parle alors de DHTML, *Dynamic HTML* [86]), des médias audio/vidéo (par l'intermédiaire de l'utilisation de *plugin*), la possibilité de définir des styles communs à plusieurs pages grâce à la norme CSS, *Cascading Style Sheet* [51], lui donnant toutes les caractéristiques d'un langage de descriptions de scènes tel que nous l'avons défini.

Bien que ce langage connaisse un succès incontestable, il présente néanmoins quelques problèmes : le premier étant lié à la gestion dynamique de la mise en forme des objets, qui ne garantit pas une mise en page précise sur tous les terminaux (le langage n'ayant pas été conçu avec cet objectif); le second lié à la nécessité d'utiliser des scripts pour l'animation, qui augmente la complexité de la création des pages; et le troisième lié à l'utilisation de *plugins* qui ne permettent pas facilement une interactivité étroite entre tous les éléments d'une page.

1.4.2 Flash

Le format des scènes multimédia le plus répandu à l'heure actuelle sur Internet est sans aucun doute le format Flash [85] de la société Adobe Macromedia [90]. De nombreux sites Internet utilisent du contenu Flash à la place du contenu HTML pour représenter une partie de la page (comme les menus, les bannières) pour rendre le site plus riche, plus dynamique, plus animé, plus interactif et avec une mise en page précise.

Le terme Flash désigne à la fois l'outil de création de contenu, le format des données internes à cet outil, non public, et le format de publication associé. Ce dernier est un format binaire dont la syntaxe

est semi-publique mais non libre. Dans cette thèse, quand nous parlerons du format Flash, par abus de langage, nous désignerons ce dernier pour lequel nous avons quelques informations.

Le format Flash a, lui aussi, subit de nombreuses évolutions dans les dernières années pour, dans les dernières versions, permettre de créer des scènes audio/vidéo. Enfin, il faut noter qu'une version restreinte du langage existe également sous le nom FlashLite [91], destinée au marché des dispositifs mobiles (téléphones, baladeurs, lecteurs DVD, appareils photos …).

1.4.3 VRML

Le langage VRML (*Virtual Reality Modeling Language*) [71], dont la deuxième version (VRML2.0 ou VRML97) a été standardisé en 1997, est également un des premiers langages de descriptions de scènes, mais orienté pour la création et la visualisation de scènes 3D. Il présente également presque toutes les caractéristiques d'un langage de descriptions de scènes complet. Il permet la description de l'organisation temporelle, spatiale, interactive et l'utilisation de média. Tout comme les autres langages, VRML a évolué depuis la première version. Une version XML, comprenant également diverses améliorations des primitives 3D, a été publiée en 2004 sous le nom *Extensible 3D* (X3D) [72]. Les limitations de ces langages sont principalement liées à la modélisation de la scène sous forme de document et à l'utilisation obligatoire d'environnement 3D.

1.4.4 BIFS

Le langage BIFS (*BInary Format for Scenes*) [39][43], partie intégrante de la norme MPEG-4, a été standardisé dans sa première version en 1999. Il s'agit d'un langage basé sur VRML qui ajoute de nombreux compléments : des primitives 2D, des primitives pour la gestion améliorée du texte, des primitives de contrôle des média plus précises, et pour une meilleure utilisation dans un contexte de diffusion, des mécanismes de compression et de *streaming*. De ce fait, BIFS est un langage de descriptions de scènes multimédia complet. Nous verrons néanmoins qu'il peut s'avérer complexe à manipuler, notamment dans les terminaux à faibles capacités de calculs.

1.4.5 SMIL

SMIL [49] (*Synchronized Multimedia Integration Language*) est un langage défini par le W3C pour décrire des scènes multimédia interactives. Il a été standardisé dans sa première version en 1998 et définit comment présenter des objets multimédia de manière synchronisée et déclarative.

Ce langage, très structuré, permet de décrire dans des parties bien distinctes de la scène, et avec précision, les caractéristiques d'animation, de synchronisation, de positionnement, de transition entre les différents objets multimédia.

Le langage SMIL est un langage de descriptions de scènes multimédia complet. Son principal avantage est l'intégration avec les langages et l'écosystème XML. Il offre entre autre la possibilité de présenter du contenu HTML animé, à la DHTML, sans utiliser de code ECMAScript. Un inconvénient de ce langage est la représentation de la scène sous forme de document qui limite les scénarios possibles de diffusion.

1.4.6 SVG

Le langage SVG (*Scalable Vector Graphics*) [50] a été également standardisé par le W3C, dans sa première version 1.0, en 2001. Il s'agit d'un langage XML permettant de décrire des objets graphiques vectoriels animés et interactifs. Il se base sur le langage SMIL pour la gestion temporelle et l'animation. Dans la version 1.2 en cours de définition, il offre également la possibilité d'intégrer des objets audio/vidéo, en faisant un langage de descriptions de scènes complet alternatif à SMIL. Ces principales caractéristiques sont, également, l'intégration avec les autres standards du W3C tels que CSS et DOM [52]. Nous présenterons dans ce document, les problèmes liés à l'efficacité de la lecture de contenus SVG, notamment dans des environnements contraints, dans le cadre de la définition de la version Tiny de ce standard.

1.4.7 LASeR

Le langage LASeR (*Lightweight Application Scene Representation*) [02][12][46] est le dernier langage de descriptions de scènes standardisé dans la norme MPEG-4. Il se base à la fois sur le langage SVG Tiny 1.2 pour la description des objets graphiques, sur le langage SMIL pour l'animation, la gestion du temps et celle des média audio-visuels, et hérite des mécanismes de *streaming* et de compression du langage BIFS.

Sa principale caractéristique est qu'il a été conçu pour les terminaux à faible capacité de calcul et de mémoire, comme les terminaux mobiles, ce qui implique qu'il est limité en fonctionnalité par rapport à d'autres langages (pas de capacités 3D) comme BIFS, mais qu'il est plus efficace dans d'autres domaines (algorithme de composition), comme nous le décrivons dans ce document.

1.4.8 CDF

La spécification CDF (*Compound Document Format*) [56], en cours de standardisation au sein du W3C, a pour but de définir les méthodes pour combiner les standards actuels du W3C (XHTML, XForm [53], SVG, SMIL, CSS) afin de fournir un langage de descriptions de scènes riche et complet basé sur des langages XML existants. Elle se concentre sur les aspects interactivité (comment interagissent deux sous ensembles de la scène exprimés dans des langages différents), et sur les organisations spatiale et visuelle (comment obtenir une cohérence spatiale et visuelle). Le but de ce

langage, à terme, est de permettre la définition de page *web* qui ne souffrent pas des limitations imposées par l'utilisation de *plugins*.

1.5 Principes piliers des descriptions de scènes

Les paragraphes précédents ont présenté rapidement quelques langages de descriptions de scènes. Les chapitres suivants présenteront les aspects liés à la compression, à l'animation, à l'interactivité et aux traitements divers qui peuvent être appliqués aux descriptions de scènes. Auparavant, afin de faciliter la compréhension de ces chapitres, nous détaillons, dans la suite de celui-ci, les piliers des langages de descriptions que nous avons énoncés précédemment.

Nous avons défini les langages de descriptions de scènes selon quatre piliers, quatre axes : l'axe de l'organisation spatiale, l'axe de l'organisation temporelle, l'axe de l'interactivité, et l'axe selon lequel le lien entre les médias et la scène est décrit. Nous décrivons ici les principes des trois premiers axes communs à la plupart des langages de descriptions de scènes. Pour le quatrième axe, nous nous intéresserons en particulier au lien entre la scène et un type de média audio visuel particulier que sont les objets graphiques vectoriels, qui sont utilisés également dans de nombreux langages (BIFS, Flash, SVG).

1.5.1 Principes de l'organisation spatiale

Comme nous l'avons indiqué précédemment, une description de scène décrit les informations nécessaires au lecteur multimédia pour positionner, sur un dispositif d'affichage, les différents éléments visuels qui composent la scène.

Cette description consiste à :

- indiquer ou définir une fenêtre de visualisation, dans laquelle le résultat de la composition de la scène est présenté. Dans un système non fenêtré, la fenêtre de visualisation correspond à l'écran tout entier;
- indiquer, si le positionnement dans cette fenêtre se déroule dans un environnement à deux ou à trois dimensions;
- indiquer, pour chaque objet, ou groupe d'objets, son positionnement.

L'organisation spatiale des objets peut être également modifiée par les paramètres de la fenêtre de visualisation. Si ceux-ci indiquent une taille de visualisation, les objets en dehors ne seront pas visibles. On parle de *clipping* dans les environnements 2D ou de *frustum culling* dans les environnements 3D. Par ailleurs, les paramètres de la fenêtre de visualisation peuvent indiquer que le rapport largeur/hauteur du contenu doit ou ne doit pas être modifié si la fenêtre est redimensionnée (action utilisateur, changement de terminal), impliquant des transformations sur le contenu. Certains langages (comme SVG) permettent également de préciser la relation entre les unités utilisées dans le

positionnement des objets et celles utilisées pour produire un rendu (par exemple pour des imprimantes) avec une résolution précise.

Certains langages permettent de spécifier une organisation spatiale complexe, liant des environnements 2D et 3D dans un même contenu, comme le montre la Figure 1.3 tirée de la norme MPEG-4.

Figure 1.3 – Scène MPEG-4 BIFS mélangeant contenus 2D et 3D

Dans les environnements 2D, les objets sont soit positionnés en donnant des coordonnées explicites : "le coin supérieur gauche de la vidéo se trouve à la position (100, 50)", soit positionnés relativement les uns aux autres : "la vidéo est alignée à gauche avec le bord de la fenêtre de visualisation". Dans le cas où les objets sont positionnés explicitement, la description de scène définit un repère, dont le centre est souvent le coin supérieur gauche de la fenêtre de visualisation, l'axe des abscisses orienté vers la droite et l'axe des ordonnées vers le bas. Certains langages, comme SMIL, permettent également de placer des objets dans des régions (zones rectangulaires).

Dans les environnements 3D, le positionnement des objets est souvent donné explicitement par des triplets de coordonnées. La coordonnée en z détermine la profondeur sachant que le plan z=0 correspond au plan de visualisation et que l'axe est orienté dans la direction s'éloignant de l'utilisateur. Dans les environnements 2D où plusieurs objets se trouvent à la même position (on parle de 2D ½), l'affichage obéit à l'algorithme du peintre. L'algorithme du peintre consiste à afficher les objets les uns par-dessus les autres, en suivant un ordre précis, comme un peintre dessinerait un tableau en commençant par le fond, puis en peignant les personnages sur le fond. L'ordre d'affichage des objets dans une scène est déterminé soit par l'ordre du document (comme en SVG), c'est-à-dire par le parcours en profondeur de l'arbre de scène, ou enfin par un ordre de profondeur explicite, souvent appelé *z-order*. Les langages BIFS et Flash permettent d'indiquer un ordre explicite d'affichage des objets dans un environnement 2D.

Enfin, les langages de descriptions de scènes permettent d'appliquer des transformations de positionnement aux objets visuels. Ces transformations sont en général les translations, rotations et

homothéties qui peuvent être appliquées à un ou plusieurs éléments de l'arbre de scène. Un intérêt de la représentation arborescente des scènes provient en effet du fait que l'on peut positionner plusieurs objets ensemble, en utilisant une seule matrice de transformation.

1.5.2 Principes de l'organisation temporelle

Pour assurer le bon déroulement temporel d'une scène non statique ou incluant des éléments audio-visuels, une description de scène doit indiquer au lecteur multimédia plusieurs informations :

- le démarrage et l'arrêt des animations et éléments audio-visuels;
- la partie de l'animation ou de l'objet audio-visuel (par exemple, un chapitre) qui doit être jouée;
- la vitesse de lecture et les contraintes de synchronisation entre les différents éléments.

Nous présentons ici les principes généraux qui permettent de comprendre comment les langages existants expriment ces trois types d'informations. Nous illustrons ensuite comment le langage SMIL [49] les exprime précisément dans ce qu'on appelle le modèle de temps SMIL.

1.5.2.1 Principes

1.5.2.1.1 Bases de temps

Pour décrire les instants de visualisation des médias (démarrage et arrêt), de même que pour l'agencement spatial, deux méthodes sont en général utilisées.

Dans la première méthode, la scène décrit l'enchaînement temporel des éléments de la présentation les uns par rapport aux autres par des évènements du type "la vidéo V démarre au chargement de la scène" ou "l'animation X s'arrête quand la vidéo V s'arrête", ou "le son S commence à jouer quand la vidéo V s'arrête". Dans ces exemples, les instants peuvent être résolus de manière déterministe dès le début de la scène. Il suffit pour cela de déterminer l'instant du chargement de la scène et la durée des médias. Dans une scène interactive, les évènements utilisés pour déclencher la lecture ou l'arrêt d'un objet média peuvent être des évènements utilisateurs. Dans ce cas, il n'est pas possible de déterminer à l'avance l'instant de début ou de fin d'un objet audio-visuel. Cette caractéristique permet une forte interactivité mais pose des problèmes si on souhaite transmettre ce type de contenu interactif sur des réseaux, notamment des problèmes de temps d'attente. Le lecteur intéressé par ces aspects pourra consulter les travaux de R. Grigoras [22] qui tente de modéliser le comportement de l'utilisateur pour précharger les contenus interactifs et réduire la latence au moment d'une action utilisateur.

La seconde méthode consiste à utiliser des valeurs numériques pour exprimer les instants de début, de fin ou les durées de visualisation des objets audio-visuels. Un exemple de déroulement temporel d'une scène, dont les instants de début, de fin ou de durée sont des valeurs numériques, est décrit dans la

Figure 1.4. On trouve par exemple "la vidéo 1 démarre à 1 seconde et se termine à 3 secondes" ou "l'audio 1 débute à 1 seconde et dure 2 secondes". Ces valeurs numériques sont exprimées par rapport à un instant T=0. Si cet instant est le même pour tous les objets animés ou audio-visuels, on dit qu'il partage la même base de temps (*timeline* en anglais), on assure un démarrage ou arrêt synchronisé des objets utilisant une même valeur numérique. En particulier, quand cet instant T=0 coïncide avec le début de la scène, on dira que la valeur numérique est exprimée dans la base de temps de la scène et on parlera pour ces valeurs numériques de temps de scène.

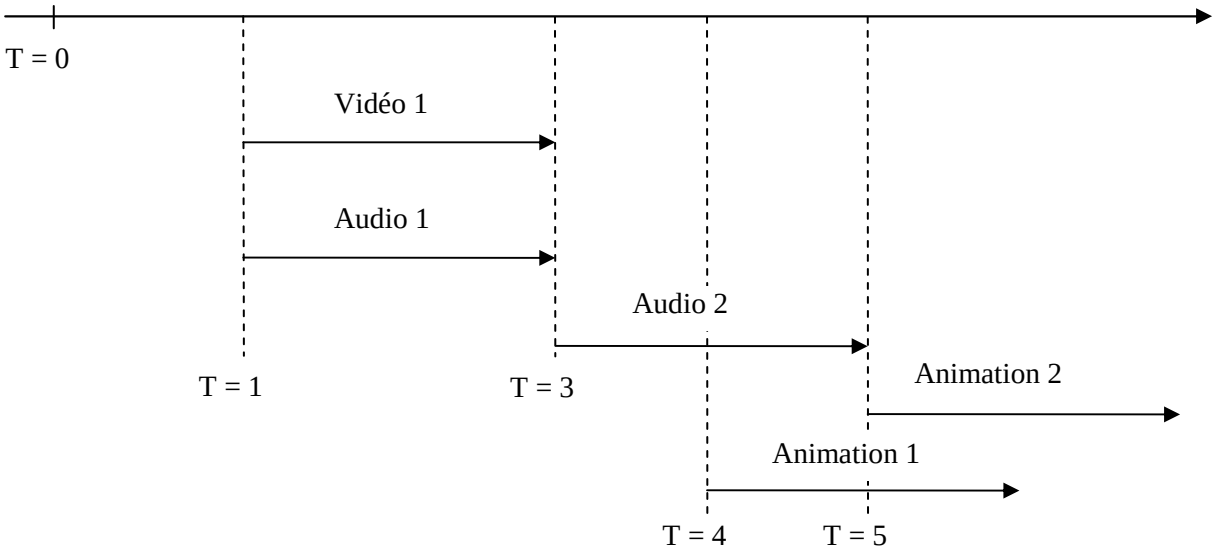


Figure 1.4 – Exemple de déroulement temporel d'une scène utilisant une base de temps

Il existe un cas très intéressant et fréquent où l'instant T=0, servant d'origine de la base de temps à un sous ensemble de la scène, ne correspond pas au début de la scène. Il s'agit de l'utilisation de *sprite* ou de *widgets* animés. La notion de *sprite*, quelque fois traduit par fantôme, est très utilisée dans les jeux vidéo pour désigner un objet animé présent à plusieurs endroits dans le jeu, quelques fois en même temps. La notion *widget*, de manière similaire, désigne un composant graphique interactif, dans une interface graphique, qui peut également être positionné à plusieurs endroits dans l'interface et animé à des instants différents. Quand de tels objets sont décrits dans une scène, il faut pouvoir décrire les instants relatifs à l'animation indépendamment du temps de scène. Pour décrire un instant dans l'animation du *sprite*, on recourt alors à l'utilisation d'une base de temps différente : la base de temps du *sprite*. On peut en fait assimiler le comportement temporel d'un *sprite* à celui d'un objet audio-visuel, il doit être démarré et arrêté à des instants précis comme s'il s'agissait d'un objet audio-visuel.

Ainsi dans le cas de scène utilisant des *sprites* (ou des éléments audio visuels) répétés plusieurs fois dans la scène et ne démarrant pas au même instant, la scène contient plusieurs bases de temps.

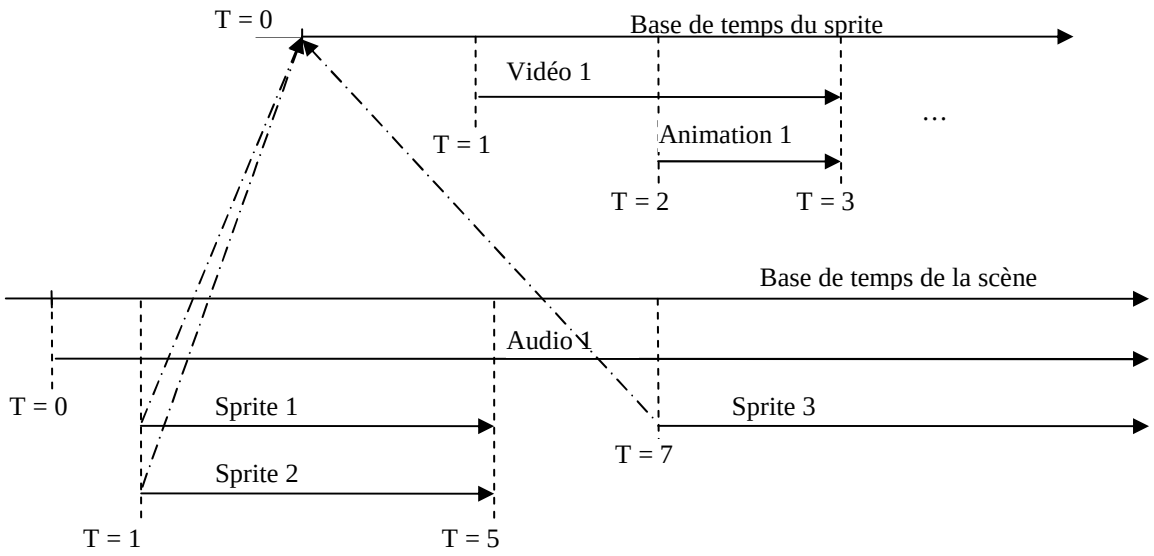


Figure 1.5 – Exemple de déroulement temporel d'une scène utilisant plusieurs bases de temps

De manière générale, la description de scène indique comment relier les bases de temps entre elles, en faisant coïncider l'origine de l'une avec un instant (origine ou pas) d'une autre, comme on peut le voir dans la Figure 1.5. On peut aboutir ainsi à une arborescence de bases de temps : la scène utilise sa propre base de temps, mais contient des *sprites* qui ont leur propre base de temps qui contiennent également d'autres *sprites* … On remarque ici un autre intérêt à la représentation arborescente de scène décrite précédemment.

1.5.2.1.2 Synchronisation

La synchronisation, selon le dictionnaire Larousse, est l'action de coordonner plusieurs opérations entre elles en fonction du temps. Cette notion de synchronisation intervient dans la présentation d'une scène multimédia, car la présentation des morceaux d'objets audio-visuels, dit continus ou à caractère temporel, doit être coordonnée en fonction des souhaits du créateur de contenu.

Pour indiquer la synchronisation entre deux objets, il n'est pas suffisant de faire coïncider l'origine des bases de temps de ces deux objets, il faut également qu'ils aient la même cadence. En d'autres termes, ce n'est pas parce que deux objets commencent à jouer à T = 10, dans la même base de temps, et qu'ils durent 10 secondes, qu'ils termineront leur lecture à T = 20. En effet, il faut prendre en compte les contraintes réseaux, les contraintes de performances qui peuvent affecter la lecture et retarder un média ou un autre. Par exemple, si deux objets audio (A) et vidéo (V) sont issus de deux serveurs/réseaux différents, que l'objet V subit des retards dans sa diffusion, une lecture synchrone nécessite d'agir soit en mettant en pause la lecture de l'objet A non retardé, soit en faisant en sorte que lors de la reprise de la lecture de l'objet retardé V, ce dernier reprenne au même instant que l'objet A. La plupart des lecteurs audio-visuels traditionnels savent en général traiter ces cas. Dans le cas de

lecteurs de contenu multimédia, il faut ajouter la prise en compte du fait que plusieurs bases de temps peuvent coexister dans le contenu.

Les langages de descriptions de scènes permettent donc d'indiquer explicitement les objets que l'auteur souhaite synchroniser. Pour cela, la plupart des langages permettent d'indiquer, pour chaque objet à caractère temporel, une base de temps de référence. Deux objets ayant la même base de temps de référence seront considérés comme synchronisés. Si leur base de temps de référence prend du retard, alors leurs bases de temps seront ajustées de manière à garder une bonne synchronisation. Certains langages, comme SMIL, permettent en plus d'indiquer une précision dans la synchronisation, à savoir la dérive maximale entre les bases de temps à partir de laquelle un ajustement sera nécessaire.

1.5.2.1.3 Temps média

Tout comme les *sprites*, les objets audio-visuels peuvent posséder leur propre base de temps. On parle de "temps média" pour désigner le temps dans la base de temps associé à un média. Il est parfois nécessaire dans des scènes multimédia complexes d'indiquer qu'on ne souhaite pas jouer un objet audio-visuel en entier mais qu'on souhaite démarrer la lecture d'un objet audio-visuel à l'instant T_m dans la base de temps du média. De même pour l'arrêt de la lecture d'un média, on peut souhaiter que celui-ci s'arrête non pas à sa fin 'naturelle' mais quand il a atteint un certain instant dans la base de temps du média. C'est le cas par exemple quand on manipule un DVD et que l'on souhaite voir un chapitre en particulier plutôt que le film tout entier.

Ainsi, les langages de descriptions de scènes permettent en général d'indiquer les temps média correspondant au début et à la fin de la lecture d'un média. De plus, certains langages, comme BIFS ou Flash, permettent à la scène d'accéder, quand le mécanisme de diffusion le permet, au temps média en cours de lecture, afin d'afficher par exemple, le temps écoulé, le temps restant, etc.

1.5.2.2 Transmission et reconstruction du temps de scène

Nous avons décrit les principes de la partie temporelle de la description de scène permettant à un lecteur multimédia de présenter des scènes en respectant les contraintes de synchronisation. Cependant, il est important de remarquer que la transmission des descriptions de scènes sur les réseaux de télécommunications introduit des délais qui peuvent poser des problèmes au niveau du lecteur de contenu dans le domaine de la gestion du temps de scène.

Le premier problème est lié au téléchargement de scènes représentées sous forme de document. Le téléchargement du document entier peut, si le document est volumineux, introduire un délai important entre les moments où les premier et dernier octets du document sont reçus. Dans certains cas, on souhaite présenter à l'utilisateur les fragments de la scène qui ont déjà été reçus, pour lui donner un aperçu. Les premiers fragments peuvent correspondre à un message d'attente lui indiquant que le contenu est en cours de téléchargement et éventuellement lui affichant une barre de progression du

téléchargement. Dans ce cas, le problème qui est posé au lecteur de contenu est de savoir si l'instant de début de présentation du premier fragment reçu correspond au démarrage de la base de temps de la scène ou si le démarrage de celle-ci doit se faire quand le document entier est reçu. Le résultat peut en effet être différent.

Par exemple, si un document est reçu sous la forme de 3 fragments reçus aux instants 0, 2 et 5 et que le dernier fragment contient une animation devant démarrer à l'instant 1 et durer 2 secondes; au moment où le dernier fragment est reçu, si la base de temps de la scène a été démarrée à la réception du premier fragment, alors l'animation atteindra directement son état final sans réellement être jouée. En revanche, si le contenu est joué à partir d'un fichier local, sous la forme d'un seul fragment, le comportement sera différent car l'animation sera jouée entièrement. Le langage SVG Tiny 1.2 définit l'attribut `timelineBegin` pour permettre à l'auteur de spécifier quel comportement il souhaite. Le langage Flash offre également des primitives permettant l'un ou l'autre des comportements.

Une solution complémentaire serait de contrôler exactement les temps de réception de chaque fragment, c'est-à-dire d'utiliser une solution de *streaming* de descriptions de scènes. Nos travaux dans ce domaine seront présentés au chapitre 6.

Un second problème de gestion du temps de scène, lié au mode de diffusion de la scène, se produit si un flux de scène est diffusé en mode non connecté. C'est le cas par exemple quand les technologies type *multicast* ou diffusion DVB sont utilisées. Chaque client se connecte, à des instants différents, à une session dans laquelle il reçoit le même flux de descriptions de scènes. Si la scène contient des animations, deux scénarios peuvent être souhaités par l'auteur :

- quand un client se connecte, les animations démarrent immédiatement, et à partir du début, c'est le cas d'une animation de bienvenue;
- ou quand un client se connecte, les animations se synchronisent sur l'état courant chez tous les autres clients, c'est le cas d'un décompte pour une application de vote qui aurait commencé avant que le client ne se connecte.

Enfin, on peut tout à fait envisager des scénarios mixtes où l'utilisateur visualise un écran de bienvenue à la connexion puis rejoint les autres clients dans le même état. Il est donc nécessaire pour ce type de scénario que l'auteur indique si une animation doit démarrer quand elle est reçue ou à un instant précis (éventuellement dans le passé); et que le serveur indique le temps commun à tous les clients au moment où le client se connecte. Cette indication peut être placée au niveau de la couche transport ou au niveau de la scène si le protocole de transport ne le permet pas, comme le définit le langage LASeR dans la commande `RefreshScene`.

1.5.2.3 Modèle de temps SMIL

La norme *SMIL Timing* définit un ensemble d'éléments et d'attributs XML pour décrire les aspects temporels d'une présentation multimédia. Elle est intéressante à présenter ici car elle illustre la plupart des points précédents.

Dans la norme SMIL, une présentation multimédia peut être vue comme un arbre dont certains nœuds sont des conteneurs temporels et dont les nœuds feuilles sont des éléments temporels. Il y a trois types de conteneurs temporels :

- l'élément groupant `par` qui indique que le déroulement de ses enfants s'effectue en parallèle;
- l'élément groupant `seq` qui indique que le déroulement de ses enfants s'effectue consécutivement dans l'ordre du document;
- et l'élément groupant `excl` qui indique que le déroulement de ses enfants s'effectue alternativement sans ordre prédéfini.

A chaque conteneur temporel est associée une base de temps. L'élément racine de la scène définit la base de temps de la scène. La base de temps d'un élément feuille est celle de l'ancêtre le plus proche en remontant l'arbre qui est un conteneur temporel. De cette manière, par défaut, la base de temps au sein d'un groupe d'éléments issus d'un même conteneur temporel est la même, assurant ainsi la synchronisation.

La norme *SMIL Timing* permet ensuite à tout élément ou conteneur temporel d'être indépendant de la base de temps de son ancêtre conteneur temporel. Ces scénarios nécessitent l'utilisation des attributs `syncBehavior`, `syncTolerance` et `syncMaster`. Le Code 1.1 décrit comment démarrer, sur un clic, la lecture synchronisée d'une vidéo et d'une audio, indépendamment du déroulement de la scène. Dans cet exemple, la base de temps de référence est indiquée par l'attribut `syncMaster` comme étant la base de temps du flux audio.

```
<par syncBehavior="independant" begin="click">
  <video src="mavideo.avi"   begin="0" syncBehavior="locked"/>
  <audio src="mamusique.mp3" begin="0" syncBehavior="locked"
                                       syncMaster="true"/>
</par>
```

Code 1.1 – Exemple de synchronisation audio-vidéo selon la norme SMIL

SMIL Timing définit également certains attributs pour indiquer les propriétés temporelles de chaque élément temporel (groupant ou feuille). Chaque élément possède des instants de début, spécifiés par l'attribut `begin`, ou de fin, spécifié par l'attribut `end`. Il possède également une durée dite simple, spécifié par l'attribut `dur`, et une durée dite d'activité, déduite à partir de sa durée simple, des instants de fin, du nombre ou de la durée de répétition, respectivement indiqués par les attributs `repeatCount`, `repeatDur`, `min`, `max` et `fill`.

Dans le modèle SMIL, un élément temporel est considéré actif pendant certains intervalles de temps. Ces intervalles de temps sont construits en prenant une des valeurs de l'attribut `begin` comme début de l'intervalle, et cette même valeur à laquelle on ajoute la durée d'activité, comme fin de l'intervalle. L'élément est considéré comme inactif en dehors de ces intervalles. Cependant, deux intervalles construits en prenant deux valeurs consécutives de l'attribut `begin` peuvent se chevaucher. Dans ce cas, l'attribut `restart` indique s'il faut fusionner ces deux intervalles en un seul, s'il faut conserver deux intervalles distincts, ou s'il faut ignorer le second intervalle.

Nous verrons dans le chapitre 2 que ces attributs sont à la base de l'animation SVG et SMIL. Mais on peut retenir de cette brève description que la gestion des éléments temporels selon le modèle SMIL est une tâche complexe puisqu'il faut potentiellement gérer plusieurs bases de temps et, pour chacune, déterminer les périodes d'activité de chaque élément.

1.5.3 Interactivité minimale : la navigation

Nous détaillerons, dans le chapitre 4, les principes régissant l'interactivité dans les langages de descriptions de scènes. Cependant, il nous semble intéressant de faire ressortir ici la caractéristique minimale, héritée du langage HTML, qui se retrouve dans la plupart des descriptions de scènes concernant l'interactivité. Il s'agit de la capacité de pouvoir naviguer entre les contenus multimédia. La navigation entre scènes indépendantes est, en effet, possible dans tous les langages décrits précédemment (Flash, SVG, BIFS, SMIL, VRML), grâce à l'utilisation de liens internes ou externes entre scènes.

1.5.4 Eléments audio visuels particuliers : les objets graphiques vectoriels

Selon notre définition, un langage de descriptions de scènes doit pouvoir décrire les média audio-visuels utilisés dans la scène. Cette possibilité peut être plus ou moins indépendante du positionnement spatial ou temporel des éléments. Par exemple, le standard MPEG-4 décrit séparément les deux aspects : les caractéristiques spécifiques aux éléments audio visuels utilisés (type, codage, …) et leurs positionnements spatial et temporel. Cette séparation est réalisée au moyen de la notion d'objet média. On peut toutefois remarquer que pour les objets graphiques vectoriels, de nombreux langages (notamment Flash, SVG et BIFS), permettent la définition d'objets graphiques vectoriels directement intégrés dans la description de la scène. Ainsi, pour afficher un rectangle, il n'est pas nécessaire de passer par un objet média de type graphique vectoriel extérieur à la scène.

Nous présentons ici deux grandes méthodes de représentation des objets graphiques utilisés dans les langages de descriptions de scènes : la représentation en carte planaire et la représentation sous forme de contours. Nous verrons dans les chapitres suivants que le choix d'une représentation a des conséquences sur les méthodes de compression, composition et animation.

1.5.4.1 Représentation par carte planaire

La représentation d'objets graphiques 2D sous forme de carte planaire est une représentation classique décrite dans [23] et notamment utilisée par le format Flash. Un objet graphique vectoriel est représenté par un seul nœud dans l'arbre de scène décrivant un ensemble de tracés (segments de droites, courbes de Bézier, etc.), dont on décrit la couleur et l'épaisseur, et pour lesquels on indique les couleurs à droite et à gauche. La Figure 1.6 tirée de la spécification Flash illustre cette représentation.

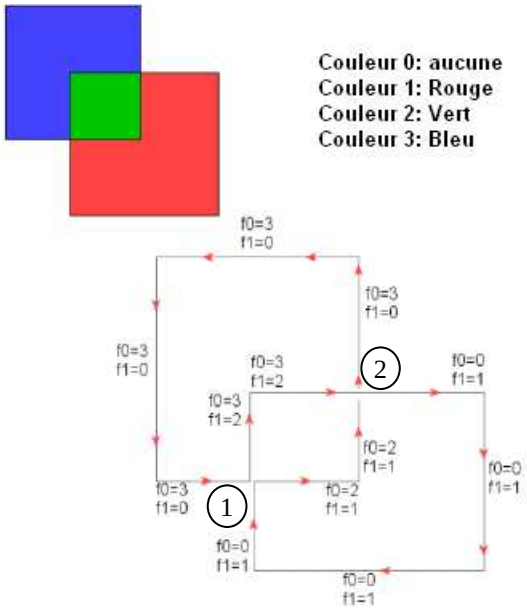


Figure 1.6 – Représentation d'une carte planaire (extrait de la spécification Flash)

La figure décrit un objet vectoriel (en haut à gauche) composé de 3 formes géométriques accolées et de couleurs différentes (rouge, vert, bleu). La partie basse indique une manière de représenter cet objet sous la forme d'une carte planaire. La flèche sur chaque tracé indique le sens de parcours du tracé. La valeur f_0 indique la couleur à gauche et f_1 la couleur à droite.

Cette représentation présente l'avantage d'être adaptée au rendu par ligne de type « scan line ». On peut par exemple, après linéarisation des courbes de Bézier, trier les segments de droites selon la coordonnée en Y puis selon la coordonnée en X, parcourir l'écran ligne par ligne et à chaque segment rencontré, changer la couleur de remplissage des pixels restants sur la ligne.

La particularité de cette représentation est qu'elle permet de définir un ensemble de traits comme un objet vectoriel formant un tout. Il n'est notamment pas possible d'interagir avec la forme verte car elle n'existe pas en tant qu'objet mais en tant que suite de tracés mélangés avec les autres tracés des autres formes géométriques. Nous verrons dans le chapitre 8 que cela impacte également l'algorithme de composition de ce type de contenu.

1.5.4.2 Représentation par contours

La représentation d'objets graphiques par contours est utilisée par beaucoup de formats notamment par les formats SVG et BIFS. Les objets graphiques sont dans ce cas représentés par une suite de primitives de tracés de contours. Il existe deux types de contours : les contours fermés (les premier et dernier points du contour coïncident), et les contours ouverts. Les contours fermés peuvent être remplis ou non. Chaque contour possède ainsi une propriété indiquant la couleur et l'épaisseur du trait, et éventuellement la couleur de remplissage. Les propriétés géométriques d'un contour sont indiquées par une liste de points et de commandes de tracés (segments de droite, courbes de Bézier cubiques ou quadratiques, arcs elliptiques). La Figure 1.7 montre un personnage de dessin animé relativement complexe composé de nombreux contours fermés et ouverts, de couleurs différentes. La Figure 1.8 montre la décomposition de ce personnage en contours (fermés et ouverts) éclatés.

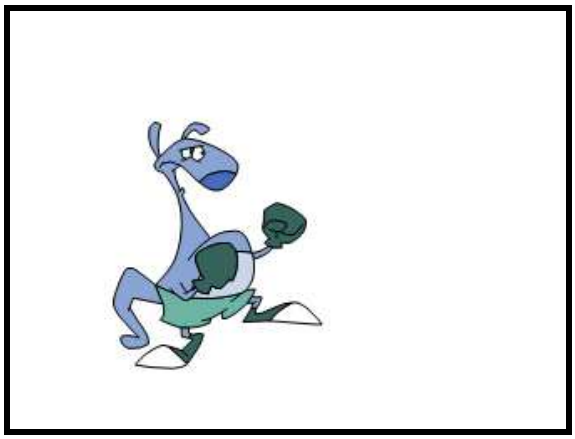


Figure 1.7 – Objet graphique vectoriel composite, formé de plusieurs contours, représentant un personnage de dessin animé

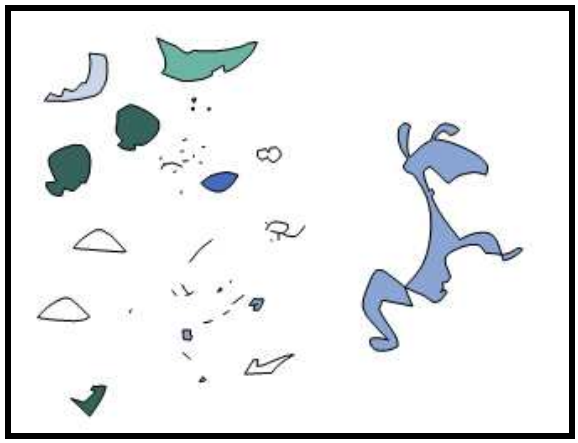


Figure 1.8 – Découpage d'un objet graphique vectoriel en contours fermés et ouverts

Dans cette représentation, les différents contours sont représentés par des nœuds dans l'arbre de scène, éventuellement regroupés ensemble, dans un sous arbre de l'arbre de scène, pour indiquer qu'ils participent à un même objet sémantique, ici le kangourou.

La particularité de cette représentation est qu'elle utilise des formes simples, certaines ayant un sens sémantique (les poings ou les yeux du personnage), pour former des objets composites plus complexes (le personnage). Cette représentation n'est pas idéale du point de vue de la composition de scènes car elle crée artificiellement des objets inutiles pour lesquels si on n'y prend pas garde, il faudra maintenir des informations pour l'interactivité, l'animation … Cet aspect est traité dans le chapitre 8.

1.6 Conclusion

Les descriptions de scènes représentent une méthode déclarative pour créer, diffuser et présenter des contenus multimédia riches, animés et interactifs. Il s'agit d'une alternative aux solutions programmatiques telles qu'OpenTV ou MHP. Dans ce chapitre, nous avons présenté quelques définitions et concepts importants pour comprendre les notions relatives aux descriptions de scènes. Nous avons également présentés brièvement les langages existants qui jouent un rôle clés dans ce domaine. Nous concluons ce chapitre en détaillant, pour les langages présentés précédemment, les caractéristiques principales selon les quatre piliers des langages de descriptions de scènes que nous avons énoncés.

Langage	Caractéristiques	Résultat
HTML	Organisation spatiale	Positionnement restreint à un environnement 2D. Positionnement successif des objets (texte et images principalement) sur une ligne ou dans une cellule d'un tableau. Hiérarchisation du positionnement possible suivant le modèle CSS de boîtes imbriquées. Pas de positionnement explicite (sauf avec CSS), ni de transformation matricielle.
	Organisation temporelle	Inexistante dans le langage HTML sans utilisation de langage de script.
	Capacité d'interactivité	Navigation inhérente. Interaction avancée possible en utilisant un langage de script.
	Lien avec les média audio-visuels	Permet l'inclusion d'objet média par l'intermédiaire de *plugin*. Interfaçage restreint avec script entre la scène et les médias.
Flash	Organisation spatiale	Positionnement uniquement matriciel et explicite dans un environnement 2D ½.
	Organisation temporelle	Structuration possible de la scène sous forme de flux. Possibilité de multiples bases de temps par l'utilisation de *sprite*.
	Capacité d'interactivité	Interactivité basée sur l'utilisation de bibliothèque de code ActionScript.
	Lien avec les média	Intégration et contrôle de tout type de média (audio,

	audio-visuels	vidéo, image), y compris de primitives graphiques vectorielles 2D animées.
VRML	Organisation spatiale	Positionnement uniquement explicite et matriciel selon un environnement 3D.
	Organisation temporelle	Utilisation d'une unique base de temps pour la scène. Utilisation de bases de temps indépendantes pour les médias.
	Capacité d'interactivité	La navigation entre scènes est possible. L'utilisation conjointe de capteurs d'évènements et de route permet une interactivité minimale. L'utilisation de script couplés avec une l'interface de programmation de la scène permet une interactivité augmentée.
	Lien avec les média audio-visuels	Définition de primitives graphiques 3D. Intégration sans contrôle d'objets audio et vidéo.
BIFS	Organisation spatiale	Positionnement explicite et matriciel ou par contrainte selon un environnement 2D et/ou 3D.
	Organisation temporelle	Utilisation d'une seule base de temps de scène. Possibilité de définir, d'utiliser et de contrôler des bases de temps différentes par le biais de plusieurs flux.
	Capacité d'interactivité	Capacités similaires à VRML augmentées par l'intégration d'un langage déclaratif de mises à jour de la scène.
	Lien avec les média audio-visuels	Définitions de primitives graphiques 2D et 3D. Intégration et contrôle de tout type de média.
SMIL 2.1	Organisation spatiale	Positionnement explicite des objets dans un environnement 2D hiérarchisé en régions.
	Organisation temporelle	Possibilité d'utiliser et de contrôler de multiples bases de temps.
	Capacité d'interactivité	Possibilité d'interaction simple sans utilisation de script par le biais de déclarations d'animations.
	Lien avec les média audio-visuels	Intégration et contrôle de tout type de média. Ne définit aucune primitive graphique.
SVG Tiny 1.2	Organisation spatiale	Positionnement explicite et matriciel des objets dans un environnement 2D. Positionnement implicite du texte dans une région 2D.
	Organisation temporelle	Restriction de l'organisation temporelle SMIL à une seule base de temps de scène par document. Utilisation possible de plusieurs bases de temps par le biais de plusieurs documents.
	Capacité d'interactivité	Capacité équivalente à SMIL, augmentée par la définition d'une interface de programmation de la scène pour l'utilisation de script.
	Lien avec les média audio-visuels	Définition de primitives graphiques vectorielles 2D. Intégration de tout type de média sans contrôle

		possible de la base de temps.
LASeR	Organisation spatiale	Equivalente à SVG augmentée de la possibilité de positionner les objets dans une table.
	Organisation temporelle	Equivalente à SVG avec la possibilité de lier des bases de temps indépendantes dans un même document.
	Capacité d'interactivité	Equivalente à SVG augmentée par l'utilisation possible d'un langage déclaratif de mise à jour.
	Lien avec les média audio-visuels	Equivalent à SVG augmenté de la possibilité de contrôler les bases temps média.

Chapitre 2 Descriptions de scènes et animations

2.1 Introduction

Parmi les caractéristiques principales d'un langage de descriptions de scènes, la capacité à décrire des animations, c'est-à-dire des évolutions temporelles de la scène, est primordiale pour créer des contenus multimédia attractifs.

L'animation d'une scène est une étape qui a lieu pendant l'affichage de la scène. En général, les animations ne s'appliquent pas à tous les attributs ou propriétés de la scène. Seules certaines propriétés sont animables. Certains langages, comme SVG ou Flash, décident que certaines propriétés ne sont pas animables quand une modification de cette valeur implique des modifications trop importantes d'un cycle d'affichage à l'autre.

Nous présentons dans ce chapitre les deux grands modèles existants : un modèle proche du fonctionnement de la vidéo, basé sur la notion de trame (*frame* en anglais) d'animation ; et un modèle issu du monde l'animation assistée par ordinateur (jeux et films 3D), basé sur des interpolations. Nous montrerons que ces approches peuvent être plus ou moins avantageuses selon le mode de consommation du contenu et plus ou moins adaptées au processus de création de contenu animé. Nous détaillerons également comment les langages décrits dans le chapitre 1 exploitent ces modèles.

2.2 Animation par interpolation

2.2.1 Principes

L'approche d'animation par interpolation est utilisée par les formats de descriptions de scènes VRML, X3D, BIFS, LASeR, SMIL et SVG. Le principe consiste à décrire toutes les étapes de l'animation en un seul bloc. Dès le début de l'animation, toutes les étapes intermédiaires sont connues. Dans cette approche, un module logiciel est chargé d'effectuer les interpolations et de piloter les animations en fonction du temps. Il peut y avoir une base de temps par animation ou pour plusieurs animations. En fonction de la résolution de l'horloge associée à cette base de temps, de la fréquence de rafraîchissement de l'affichage et des ressources processeur disponibles au moment de la lecture, le

module de pilotage de l'animation peut décider (ou être paramétré) du nombre d'interpolations à effectuer par seconde et donc de la fluidité ou non de l'animation.

La Figure 2.1 décrit une architecture simplifiée d'un lecteur multimédia implémentant un module d'animation de scènes par interpolation. La partie encadrée par des pointillés correspond à la partie supplémentaire par rapport à un lecteur multimédia sans capacité d'animation par interpolation. La synchronisation de la scène peut ou non être effectuée de la même manière que la synchronisation des flux média.

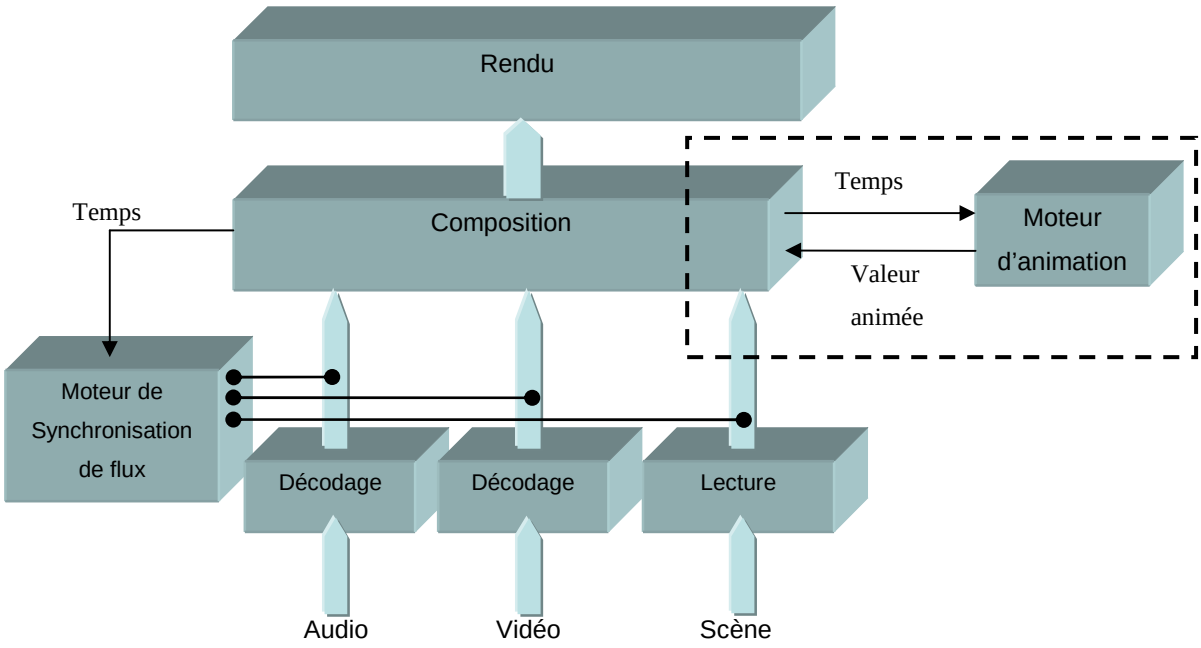


Figure 2.1 – Architecture simplifiée d'un lecteur multimédia capable de traiter des animations par interpolation

Les différences entre les langages de descriptions de scènes utilisant cette approche se situent à trois niveaux :

- dans le modèle de temps sous-jacent, c'est-à-dire la manière de déterminer le temps de l'animation et de le transmettre au moteur d'animation;
- dans les types d'interpolations possibles;
- et dans la manière de transmettre la valeur interpolée (utilisation d'évènements ou non) et de l'appliquer à la scène.

Le fait de décrire tous les états de l'animation ensemble présente l'avantage de pouvoir : utiliser des techniques efficaces de codage pour rendre la représentation compacte [24], optimiser le traitement de l'animation par des précalculs effectués à l'initialisation, et enfin de pouvoir utiliser des techniques d'interpolation avancées fluides (*spline*) [25]. Cependant, avec cette description en un bloc, il n'est pas

possible de diffuser en continue de nouveaux états à l'animation sans réinitialisation. L'aspect déterministe de l'animation peut permettre une exécution plus fluide mais rend cette approche peu maniable dans un contexte interactif, où on souhaiterait modifier l'animation en réponse à une interaction de l'utilisateur. Les animations discrètes ou mises à jour s'y prêtent plus comme nous le verrons par la suite.

2.2.2 Applicabilité

Pour cette approche, on parle également d'approche paramétrique car c'est le paramètre « temps » qui détermine le résultat de l'animation. La spécification de l'animation consiste à décrire la fonction d'animation, c'est-à-dire à donner son début et sa durée (ou sa fin) et les valeurs : sous la forme d'une fonction mathématique, continue ou non, ou d'une liste de valeurs. La Figure 2.2 illustre une animation paramétrique, continue du temps, qui décrit un mouvement de caméra de type zoom arrière.

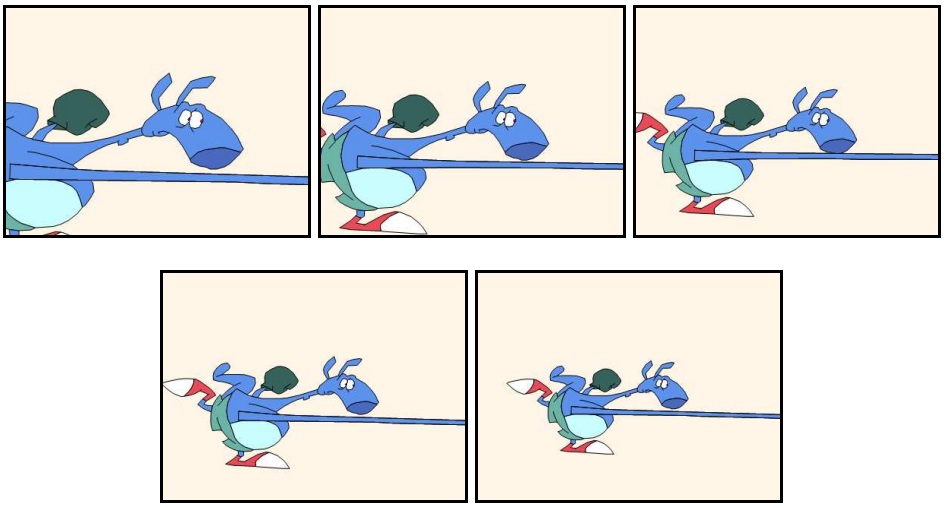


Figure 2.2 – Illustration d'une animation paramétrique : un mouvement de caméra

Cette approche est bien adaptée pour représenter des animations issues d'un processus de création de contenu informatisé, où le créateur de contenu sélectionne dans un outil auteur une valeur de départ, une valeur de fin et éventuellement quelques états intermédiaires, puis spécifie les méthodes d'interpolations (linéaire, cubique …).

2.2.3 Langages utilisant le modèle d'animation par interpolation

Nous présentons dans la suite comment certains langages existants utilisent le modèle présenté ci-dessus.

2.2.3.1 VRML/BIFS

Le modèle pour l'animation par interpolation adopté par la norme MPEG-4 BIFS est issu de VRML. Il fonctionne de la manière suivante : un nœud de l'arbre de scène, appelé `TimeSensor`, permet de contrôler les paramètres temporels de l'animation (durée, répétition) ; le résultat du traitement de ce

nœud, à chaque cycle de composition, est une valeur comprise entre 0 et 1 ; cette valeur est ensuite utilisée par un nœud chargé de calculer la valeur interpolée ; enfin la valeur interpolée est utilisée pour modifier une propriété de la scène. Le Code 2.1 décrit les caractéristiques du nœud `TimeSensor`.

```
TimeSensor {
  exposedField SFTime   cycleInterval    1
  exposedField SFBool   enabled          TRUE
  exposedField SFBool   loop             FALSE
  exposedField SFTime   startTime        0
  exposedField SFTime   stopTime         0
  eventOut     SFTime   cycleTime
  eventOut     SFFloat  fraction_changed
  eventOut     SFBool   isActive
  eventOut     SFTime   time
}
```

Code 2.1 – Définition du nœud VRML TimeSensor

Pour l'interpolation, la norme MPEG-4 BIFS définit les nœuds suivants :

`ScalarInterpolator`, `ColorInterpolator`, `CoordinateInterpolator2D`, `CoordinateInterpolator`, `CoordinateInterpolator4D`, `PositionInterpolator`, `PositionInterpolator2D`, `PositionAnimator`, `PositionAnimator2D`, `OrientationInterpolator` et `NormalInterpolator`. Le Code 2.2 donne plus de détail sur les paramètres d'un nœud d'interpolation.

```
ColorInterpolator {
  eventIn      SFFloat set_fraction
  exposedField MFFloat key
  exposedField MFColor keyValue
  eventOut     SFColor value_changed
}
```

Code 2.2 – Définition du nœud ColorInterpolator

Ces nœuds possèdent tous les propriétés `set_fraction`, `key`, `keyValue` et `value_changed`. Les types de ces propriétés sont dépendants du type de la valeur interpolée. Par exemple, le nœud `ScalarInterpolator` produit une valeur décimale alors que le nœud `CoordinateInterpolator` produit un triplet de valeurs décimales. Ce résultat est ensuite appliqué à une propriété d'un nœud visuel ou d'un nœud de transformation. Par exemple, la valeur décimale issue du nœud `ScalarInterpolator` peut être utilisée pour modifier l'épaisseur d'un contour et le triplet pour modifier les dimensions d'un cube.

Le mécanisme utilisé dans les scènes de type VRML (MPEG-4 BIFS, X3D) pour permettre aux différents nœuds de communiquer est le mécanisme des « routes ». Dans le cas précis d'une animation, deux routes signalent : d'une part, qu'un nœud `TimeSensor` communique avec un nœud d'interpolation et que, d'autre part, ce dernier communique avec le nœud cible. Le mécanisme de « route » fait partie du modèle événementiel de la norme. C'est le même mécanisme qui, par exemple,

transmet un événement de la souris à un nœud cible. Il permet notamment de définir une seule animation dont la valeur interpolée sera transmise à plusieurs nœuds cibles. Enfin, il faut noter qu'une fois la valeur interpolée transmise à un nœud cible, il n'est pas possible de revenir à la valeur avant animation. La Figure 2.3 schématise le processus d'animation VRML.

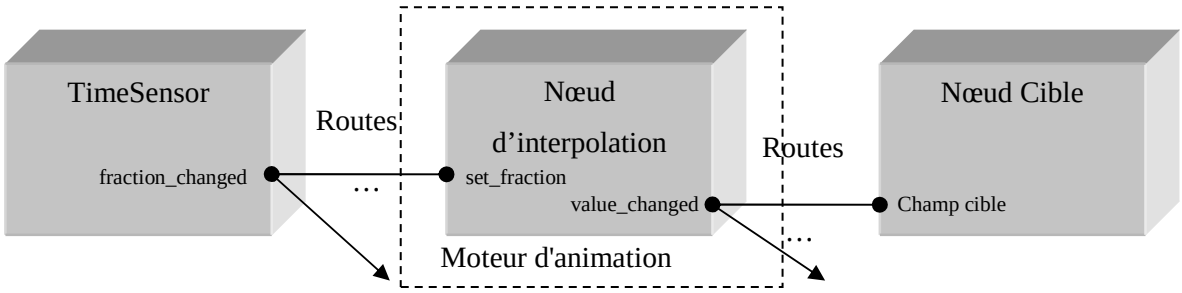


Figure 2.3 – Diagramme d'exécution d'une animation VRML/BIFS

Ce modèle est assez simple. A chaque cycle de présentation, il suffit pour le module de composition de déclencher des événements de temps issus du ou des nœuds `TimeSensor`, qui sont traités par les nœuds interpolateurs, qui génèrent à leur tour des évènements qui sont traités par le ou les nœuds cibles. Les animations sont donc bien séparées les unes des autres. Si par exemple, deux interpolateurs ont la même cible, la valeur finale sera celle obtenue en suivant la dernière route. De même, le modèle de temps lié à ce type d'animation est relativement simple. Le paramètre « temps » de l'animation est dicté par le nœud `TimeSensor` qui évolue en fonction du temps dans la base de temps dont il dépend. Synchroniser deux animations peut se faire simplement en utilisant le même nœud `TimeSensor` comme source du temps.

2.2.3.2 SMIL/SVG

Le modèle d'animation utilisé dans les standards du W3C comme SVG, issu de la norme SMIL, est également le modèle par interpolation. La norme SMIL est divisée en de nombreuses parties. La partie qui nous concerne ici est la partie *SMIL Animation*.

2.2.3.2.1 Animations SMIL

2.2.3.2.1.1 Spécifier une animation

Le modèle d'animation SMIL est fondé sur le modèle de temps SMIL présenté dans le chapitre 1. Chaque élément d'animation SMIL est un élément temporel. Quand il est actif, l'animation est effective. Quand il est inactif, l'animation n'a pas lieu. Dans la terminologie SMIL, une animation peut rester active indéfiniment. Nous verrons que c'est une contrainte importante pour l'implémentation efficace d'un lecteur de scènes animées.

Spécifier une animation dans les langages SMIL ou SVG revient à indiquer la fonction d'animation sur une durée correspondant à la durée simple, comme nous l'avons décrit dans le chapitre 1, et à

indiquer le comportement de l'animation quand elle se termine : si elle revient dans l'état initial ou si elle maintient l'état final indéfiniment. Le comportement de l'animation sur la durée active est obtenu en répétant le comportement sur la durée simple.

Pour indiquer la fonction d'animation sur la durée simple, chaque élément d'animation possède des attributs qui permettent d'indiquer des valeurs clés : celle du début (attribut `from`), celle de fin (attribut `to`), un incrément (attribut `by`) ou la liste des valeurs possibles (attribut `values`). Chaque élément indique également la nature de l'interpolation entre les différentes valeurs : discrète, linéaire, cubique, linéaire à vitesse constante … en utilisant l'attribut `calcMode`.

SMIL définit quatre éléments d'animation mais nous verrons que SVG a étendu cette liste. Ces éléments sont les suivants : `set`, `animate`, `animateMotion` et `animateColor`. Chaque élément est utilisé pour des animations d'un certain type. Les éléments `animateMotion` et `animateColor` servent respectivement à animer les déplacements d'un objet et une des couleurs associées à un objet (par exemple la couleur de remplissage ou celle du contour, car SVG utilise la représentation graphique sous forme de contour décrite dans le chapitre 1). L'élément `set` sert à effectuer des animations discrètes, sans interpolation, sur des attributs de type numérique ou littéral. Enfin, l'élément `animate` est un élément générique qui permet d'animer tout attribut animable.

Chaque élément d'animation cible un seul élément. L'animation SMIL utilise un mécanisme d'adressage des éléments cibles sur lesquels vont s'appliquer l'animation. Le mécanisme de route n'existant pas, et SMIL étant un langage XML, il réutilise les mécanismes bien connus d'adressage XML : l'utilisation d'identifiant unique, et l'utilisation de la syntaxe XPointer [58].

2.2.3.2.1.2 Appliquer et combiner des animations

La manière de spécifier les animations en SMIL est relativement proche de celle utilisée en MPEG-4 BIFS ou en VRML. On indique une liste de valeurs clés et d'instants clés ainsi que des paramètres pilotant l'interpolation. Cependant, la grande différence avec le modèle VRML réside dans la façon dont ces animations sont appliquées à la scène et la façon dont elles sont combinées.

En effet, le modèle d'animation SMIL utilise le concept de valeur de présentation. Quand une animation est active, conceptuellement, elle ne modifie pas directement la valeur d'un attribut dans l'arbre DOM. En réalité, quand une animation est active, ce n'est plus la valeur de l'arbre DOM qui est utilisée pour la présentation mais la valeur issue de l'animation ou de la combinaison de plusieurs animations, qui sont déduites de la valeur DOM. Le modèle SMIL permet de spécifier si, à la fin d'une animation, la valeur de présentation est maintenue ou si la valeur de l'arbre DOM doit être à nouveau utilisée pour présenter le document.

SMIL définit le comportement quand plusieurs animations ont pour cible le même attribut grâce à l'attribut `additive`. Selon la valeur de cet attribut, les animations se cumulent ou se remplacent. De

plus, l'ordre dans lequel les animations sont appliquées dépend de l'ordre dans lequel elles ont démarré, et si elles ont démarré en même temps, de l'ordre de déclaration dans le document. La combinaison des animations fonctionne comme suit. Chaque animation calcule sa valeur de sortie, la valeur de présentation, en fonction d'une valeur d'entrée : soit cette valeur d'entrée est indiquée dans les attributs `from` ou `values`, soit l'animation utilise la valeur sous-jacente. Pour la première animation, la valeur sous-jacente est la valeur contenue dans l'arbre DOM[2]. Pour les animations successives, la valeur sous-jacente est la valeur de présentation produite par l'animation précédente. Le résultat final, utilisé pour l'affichage, à chaque cycle de présentation, est la valeur de présentation issue de la dernière animation à s'être exécutée. Ce fonctionnement est illustré dans la Figure 2.4.

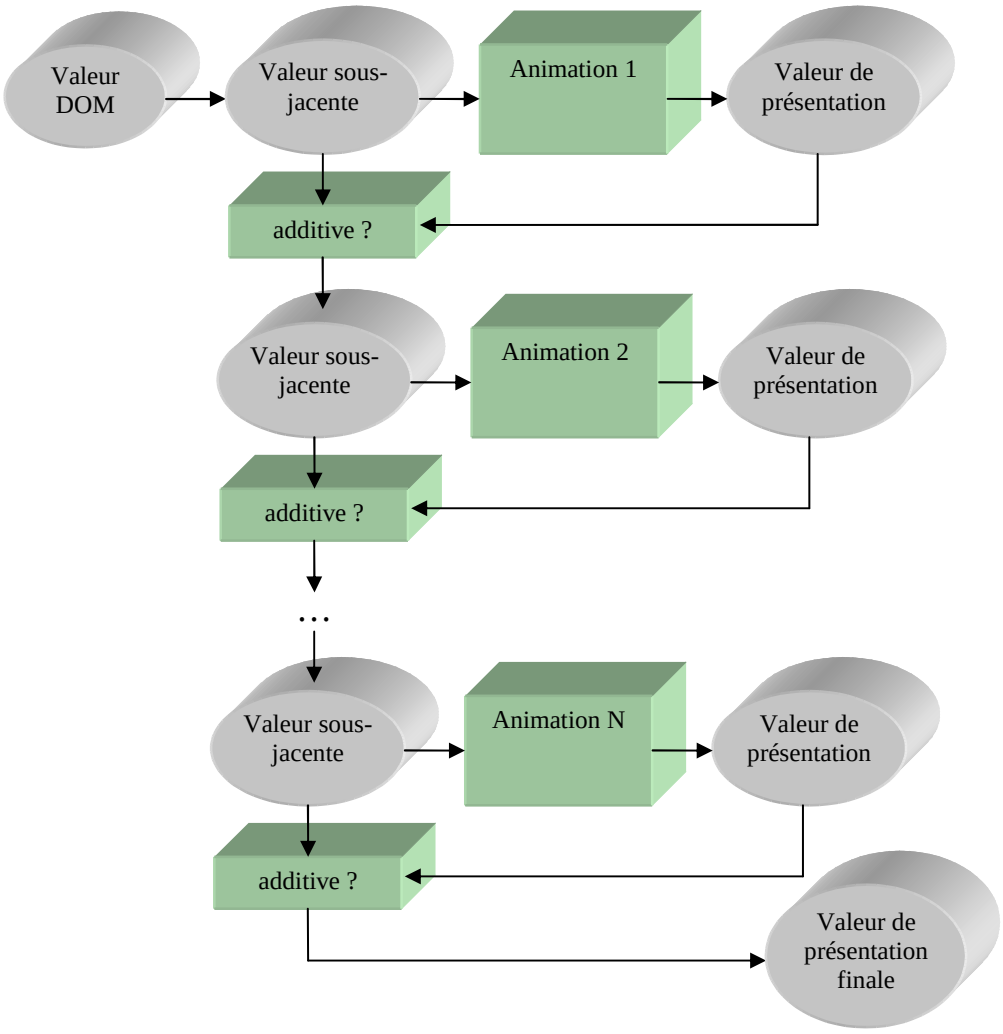


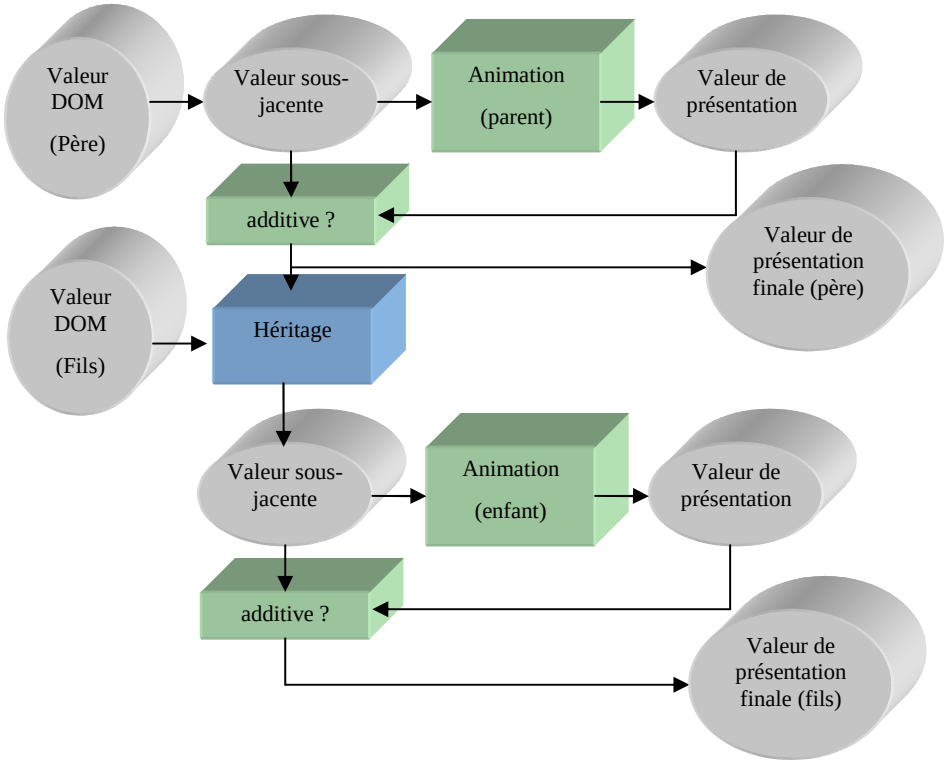
Figure 2.4 – Diagramme d'exécution de N animations SMIL s'appliquant à un seul attribut d'un même élément cible

[2] On suppose ici qu'il n'y a pas d'héritage.

2.2.3.2.1.3 Héritage et animation

Une autre différence entre le modèle d'animation SMIL et celui de la norme VRML est son interaction possible avec la norme CSS (*Cascading Style Sheet*). En effet, cette dernière définit notamment un ensemble de propriétés qui peuvent être spécifiées sur un élément groupant de l'arbre de scène mais qui doivent être transmises et éventuellement modifiées de proche en proche aux différents niveaux dans le sous-arbre issu de ce nœud. On parle d'héritage de propriétés. Si une propriété héritable n'est pas spécifiée au niveau N ou est spécifiée avec la valeur `inherit`, alors la valeur utilisée, pour les calculs d'animations et l'affichage, est la valeur issue du niveau N-1[3].

La conjonction de l'utilisation de CSS et de *SMIL Animation* sur un même arbre de scène entraîne une étape supplémentaire pendant l'animation. Il faut, en effet, avant chaque calcul d'animation, déduire la valeur sous-jacente en effectuant le mécanisme d'héritage de la racine de l'arbre jusqu'au nœud cible de l'animation. La Figure 2.5 illustre ce mécanisme dans la situation suivante. Un même attribut est animé sur 2 nœuds père/fils et la valeur de l'attribut animé du nœud fils implique un héritage de la valeur animée au niveau du parent.



[3] La valeur issue du niveau N-1 n'est pas nécessairement la valeur spécifiée au niveau N-1, certains calculs peuvent être appliquées (résolution de pourcentage …). On parle de *computed value*.

Figure 2.5 – Animation SMIL et héritage CSS

Nous verrons dans le chapitre 7 que ce fonctionnement a un impact notable sur l'implémentation d'un module d'animation SMIL.

2.2.3.2.2 Animations SVG

SVG est appelé un langage hôte pour la spécification SMIL, car il réutilise les concepts issus des normes *SMIL Timing* et *SMIL Animation*, en précisant certaines restrictions et définissant de nouveaux éléments d'animation. Parmi les nouveaux éléments d'animation, on trouve : l'élément `animateTransform`, qui indique une animation de la matrice de transformation d'un élément graphique ; l'élément `mpath` qui précise l'utilisation de l'élément `animateMotion` en indiquant un chemin, éventuellement complexe (comme une courbe de Bézier), le long duquel s'effectue le déplacement de l'objet animé. Nous avons également introduit, dans la version 1.2 de cette norme, l'élément `discard`, qui fonctionne comme l'élément `set` mais qui a pour but de signaler quand un objet peut être détruit. Nous détaillerons l'utilisation de tous ces éléments dans le cadre du dessin animé vectoriel dans le chapitre 6.

2.2.4 Synthèse sur les animations par interpolation

Nous avons vu deux méthodes pour décrire des animations se basant sur un modèle d'interpolation. Il faut retenir que ces animations sont adaptées pour la déformation, le déplacement continu d'objet, qu'elles permettent d'avoir des animations très fluides tout en gardant une représentation compacte, mais qu'elles nécessitent un moteur d'animation qui peut être coûteux en fonction des types d'animation, de la complexité de combinaison des animations. De plus, on peut conclure en disant que le modèle VRML est plus simple à traiter au niveau du lecteur (traitement du temps, combinaisons d'animations) mais moins souple que le modèle d'animation SMIL pour ce qui concerne la création de contenu, car il nécessite l'utilisation de trois primitives : deux nœuds et deux routes.

2.3 Animation par trames d'animations

2.3.1 Principes

Le modèle d'animation par trame[4] est utilisé par les langages de descriptions de scènes Flash, BIFS, LASeR et dans une certaine mesure, les langages XHTML, SMIL et SVG peuvent être étendus pour le supporter.

Le principe général consiste à décrire la scène dans son état initial et à décrire ensuite les mises à jour qui lui sont appliquées. On parle également, pour ces mises à jour, de trames d'animation. On peut distinguer plusieurs critères différenciant les formats d'animation par trames : les types de mises à jour possibles, les conséquences évènementielles, le mécanisme d'adressage pour indiquer la cible des mises à jour.

De manière générale, et quel que soit le format, on associe à chaque mise à jour une estampille temporelle qui indique le temps auquel cette mise à jour doit être appliquée à la scène. Cette association peut être faite au niveau de la scène, soit en indiquant un temps d'exécution dans la base de temps de la scène, soit en utilisant un évènement; ou au niveau du protocole de transport si les mises à jour sont transportées séparément, en utilisant l'estampille temporelle comme indicateur de l'instant d'exécution des mises à jour. La suite de la description de la scène initiale et de ses mises à jour temporelles forme dans ce cas ce qu'on appelle un flux de descriptions de scènes comme cela a été abordé au chapitre 1.

Dans cette approche, le déroulement de l'animation est piloté par l'exécution des mises à jour. Le lecteur de contenu n'a pas à déterminer de valeur d'animation entre deux mises à jour car les seules valeurs d'animations possibles sont indiquées par les mises à jour et les instants précis de l'animation sont indiqués par les estampilles temporelles.

Le pilotage de l'animation peut éventuellement être effectué par un serveur distant. Dans un scénario d'utilisation où un serveur envoie ces mises à jour, le serveur contrôle l'état de l'animation au niveau du client puisqu'il ne peut y avoir de modification chez ce dernier si le serveur n'envoie pas de mise à jour. C'est une différence majeure avec le modèle par interpolation.

[4] On peut également remplacer le mot trame par mise à jour, image, pose, photogramme …

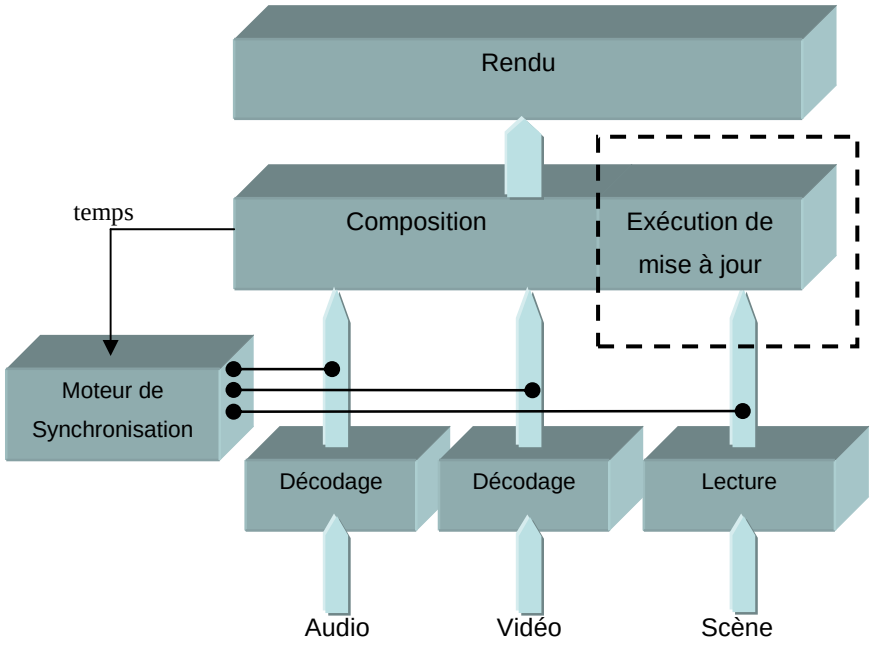


Figure 2.6 – Architecture simplifiée d'un lecteur multimédia comprenant un flux de descriptions de scènes

La Figure 2.6 décrit une architecture simplifiée de lecteur multimédia capable de traiter un flux de descriptions de scènes. La partie située dans le cadre en pointillés indique la partie spécifique au traitement du flux de descriptions de scènes. Cette approche d'animation ne nécessite pas de module logiciel au sein du lecteur de contenu qui soit chargé de traiter le temps d'animation ni de faire les calculs d'interpolation. Le lecteur doit seulement effectuer les mises à jour quand leurs estampilles arrivent à échéance. Or, ce traitement des échéances est le même que celui qui est appliqué aux flux audio ou vidéo, ce qui rend très facile l'intégration d'un moteur d'animation par mise à jour dans un lecteur audio-visuel classique.

2.3.2 Applicabilité

On utilise l'approche par trame d'animations pour décrire une animation quand il n'est pas possible de décrire les modifications de la scène comme une fonction (ou une combinaison de fonctions) continue(s) du temps. On l'utilise également lorsqu'on souhaite fixer la fréquence de rafraîchissement de l'animation. Elle est utile notamment quand, dans la description d'un objet graphique, de nouveaux objets (traits, contours) apparaissent, disparaissent ou quand des déformations non continues sont appliquées à un objet. La Figure 2.7 montre quatre images consécutives dans une animation dont la fréquence de rafraîchissement est de 12 images par seconde. On peut voir que d'une image à l'autre, les différences peuvent être très importantes, notamment sur le personnage et sur les décors : ils sont très déformés et de nombreux éléments graphiques ont disparu ou sont apparus (flash et débris suite à l'explosion…). On se rend ici compte de l'impossibilité d'une représentation paramétrique simple.

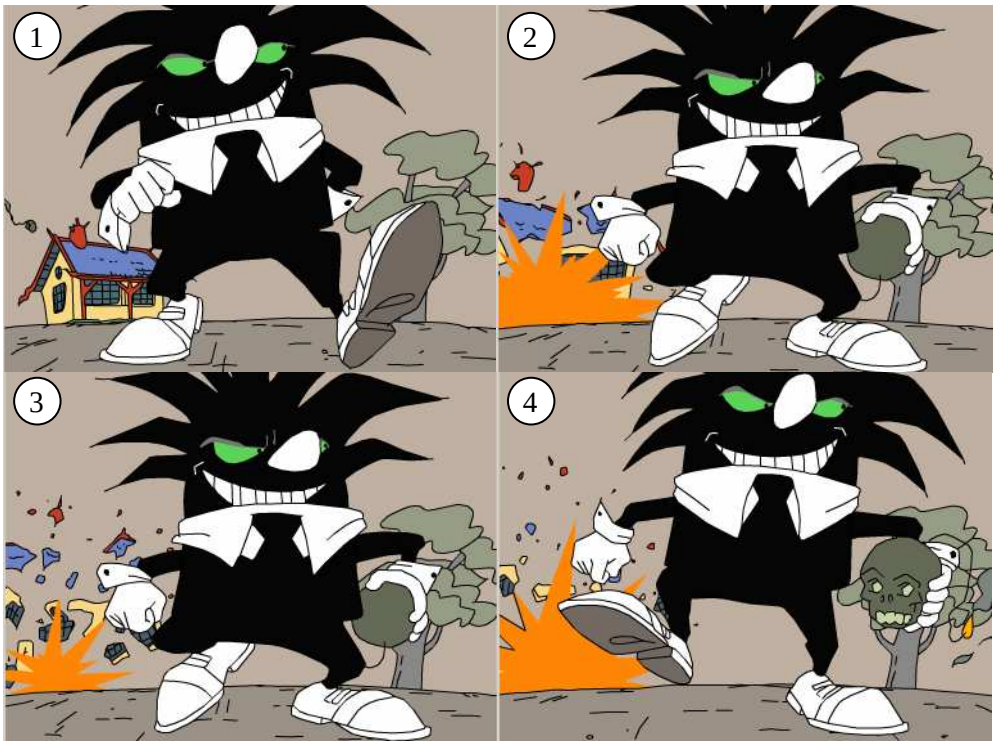


Figure 2.7 – Illustration d'une animation non continue

Dans le cadre de cette thèse, nous avons été amenés à travailler avec des sociétés de production de dessins animés [104]. Dans ces sociétés, le processus de création (détaillé dans le chapitre 6) repose sur des dessinateurs qui travaillent sur papier et dessinent trame par trame le dessin animé. Etant donné que les états successifs d'un personnage présent d'une trame à l'autre sont dessinés à partir d'une feuille blanche, il est impossible, de manière générale, de représenter la succession de ces états par une même combinaison de modifications (translation, rotation, mise à l'échelle, …) continues dans le temps. Cette approche est donc bien adaptée pour représenter des animations issues de ce type de processus de création de dessins animés vectoriels non informatisés.

2.3.3 Instances existantes et théoriques du modèle d'animation par trame

Nous présentons ici les langages qui, parmi ceux présentés dans le chapitre 1, utilisent un modèle d'animation par trame. Nous présentons également des extensions (théoriques ou en cours de standardisation) de certains autres langages dans ce domaine.

2.3.3.1 Trames Flash

Le format Flash est l'exemple type illustrant l'approche d'animation par trames. En effet, il ne permet que l'utilisation de cette approche et contrairement aux autres formats, ne permet pas la combinaison des deux approches. Bien qu'il soit possible, dans l'outil de création Flash, de créer des animations par

interpolation, le fichier résultant contiendra une suite de mises à jour approchant l'interpolation voulue.

Une trame d'animation contient les mises à jour s'appliquant à un même instant. Certaines trames peuvent éventuellement être vides, mais toutes s'exécutent à intervalles réguliers, selon la fréquence d'animation définie dans l'entête du flux ou du *sprite*.

De même que la représentation de la scène Flash est simple, les types de mises à jour sont également très limités et simples. Le format autorise les types de mise à jour suivants :

- la définition d'un nouvel objet du dictionnaire;
- le placement d'un objet du dictionnaire dans la liste d'affichage;
- la modification des propriétés d'une profondeur dans la liste d'affichage;
- le retrait d'un objet de la liste d'affichage;
- et la modification du coefficient de déformation pour certains symboles.

Le mécanisme d'adressage utilisé dans le format Flash est basé sur des identifiants numériques uniques. Chaque objet dans le dictionnaire possède un identifiant numérique unique dans le flux. De plus, chaque position dans la liste d'affichage ne peut contenir qu'un seul objet. Le couple (identifiant, profondeur) est donc unique et surtout suffisant pour adresser les besoins de types de mises à jour du langage.

Ainsi, la principale caractéristique du format Flash concernant l'animation est sa simplicité, alors qu'il permet néanmoins la création de contenu très attractif et efficace. On peut en juger par le succès de Flash sur Internet et la version du lecteur *FlashLite* qui utilise les concepts décrits dans cette partie pour le multimédia mobile.

2.3.3.2 Trames MPEG

Deux autres formats de descriptions de scènes utilisant ce modèle ont été standardisés par le groupe MPEG. Il s'agit des formats MPEG-4 BIFS et MPEG-4 LASeR. Ces formats sont deux formats binaires qui reposent sur les mêmes principes mais se différencient principalement par le format de la scène initiale et les techniques de codage. La représentation de la scène initiale est donnée sous la forme d'un arbre. Dans le cas du format BIFS, cet arbre est formé de nœuds BIFS. Dans le cas du format LASeR, l'arbre est déduit de la représentation XML d'une scène SVG. Dans les deux cas, les types de mises à jour possibles sur cet arbre sont les suivants :

- insertion d'un nouveau nœud dans l'arbre;
- suppression d'un nœud de l'arbre;
- remplacement d'un nœud de l'arbre par un nœud donné dans la mise à jour;

- remplacement de la scène;
- et modification d'une propriété d'un nœud[5].

Une différence importante avec le format Flash décrit précédemment réside dans le format de la scène initiale. Dans ces formats MPEG, la scène initiale est plus complexe car il ne s'agit plus d'une liste d'affichage mais d'un arbre. Nous verrons les avantages et inconvénients de cette représentation dans le chapitre 7. En revanche, pour ce qui concerne le mécanisme de mise à jour, les méthodes Flash et MPEG sont similaires.

Un des choix importants qui a été fait dans les normes MPEG-4 BIFS et LASeR concerne l'adressage des objets. Il ne s'agit plus, comme dans le langage Flash, d'adresser un objet dans une liste ou dans un dictionnaire mais dans un arbre complet. Nous présentons ici les deux techniques, classiques, évaluées par le groupe MPEG.

La première consiste à indiquer le chemin depuis la racine de l'arbre jusqu'au nœud cible. Cette approche est à la base du standard XPath [57] et est utilisée dans la norme MPEG-7 *Systems* [26]. Elle est très expressive car elle permet d'adresser n'importe quel endroit dans l'arbre. Elle présente également de bonnes propriétés de résistance aux erreurs. Si par exemple, une partie de l'arbre est perdue pendant la transmission, et si une mise à jour suivante concerne un des endroits qui a été perdu, l'adresse du nœud ou de la propriété cible peut toujours être interprétée. Cependant, le niveau d'interprétation dans le cas d'une scène graphique peut être limité. Si, par exemple, le nœud perdu est responsable du positionnement d'un objet graphique, les enfants, non perdus, de ce nœud seront présents dans la scène mais mal positionnés.

La seconde consiste à adresser un nœud en lui attribuant un identifiant unique dans la scène (ou une partie). C'est une approche similaire à celle de Flash et également un des principes de la norme XPointer. Bien qu'elle soit moins expressive, c'est le choix retenu pour LASeR et BIFS. Ce choix s'explique par sa faible complexité et son mode d'utilisation. LASeR et BIFS reposent, en effet, sur l'hypothèse suivante : quand une scène est créée, les nœuds qui seront animés sont connus à l'avance, lors de l'édition de la scène, et surtout sont peu nombreux. Prenons l'exemple suivant, un match de football est diffusé sous la forme d'une scène MPEG-4 et permet à l'utilisateur de choisir les statistiques qu'il souhaite voir sur le match. Ces statistiques sont mises à jour en temps réel par le diffuseur. Les éléments graphiques (texte, images, logo …) qui seront modifiés pendant le match sont

[5] Ici nous omettons les quelques différences de mise à jour d'un attribut qui existent entre les langages BIFS et LASeR, ce dernier étant plus complet de ce point de vue.

connus à l'avance, on peut donc leur attribuer un identifiant. Le choix du groupe MPEG a donc été de limiter l'expressivité du mécanisme d'adressage pour limiter la complexité et coller à des scénarios d'utilisation concrets.

2.3.3.3 Trames W3C

Nous avons vu précédemment que certains langages du W3C, comme SVG et SMIL, intègrent des primitives pour décrire des animations par interpolation. Dans cette partie, nous montrons qu'il est également possible, de plusieurs manières, de procéder à des animations par mises à jour sur des scènes utilisant ces langages.

2.3.3.3.1 Animations SMIL discrètes

Une première manière repose sur l'utilisation d'animations discrètes (image par image) telles que les langages SMIL et SVG les spécifient. Les Code 2.3, Code 2.4 et Code 2.5 illustrent cette possibilité.

```
<svg>
<text id="titre">Ceci est le titre</h1>
<text id="texte_initial" display="inline">Ceci est le texte initial</p>
<text id="texte_final" display="none">Ceci est le texte final</p>
</svg>
```

Code 2.3 – Exemple de scène initiale SVG

```
<svg>
<text id="titre">Ceci est le titre</h1>
<text id="texte_final" display="inline">Ceci est le texte final</p>
</svg>
```

Code 2.4 – Exemple de scène SVG mise à jour

Les outils qui nous intéressent font partie des outils d'animation. Il s'agit des éléments `set` et `discard`. L'élément `set` permet d'effectuer une animation discrète à un instant donné. Cette animation peut être assimilée à une mise à jour, même si elle ne modifie pas réellement le document XML comme nous l'avons vu dans la section 2.2.3.2.1. Les mises à jour, que l'élément `set` permet de modifier, affectent la valeur utilisée pour le rendu (la valeur de présentation) de certains attributs animables : des valeurs décimales, des chaînes de caractères, des identifiants …. Le Code 2.5 montre comment former une mise à jour SVG pour obtenir le résultat souhaité. L'élément `discard` fonctionne sur le même principe et permet de supprimer, à un instant donné, un élément de la scène.

```
<set begin="2" xlink:href="#texte_final" attributeName="display"
  to="inline"/>
<discard begin="2" xlink:href="#texte_initial"/>
```

Code 2.5 – Exemple de mises à jour SVG

Cette solution est envisageable en utilisant la version 1.2 de la norme SVG, qui intègre l'élément `discard`. Cette solution est également envisageable dans un format combinant les normes XHTML et SMIL dans le cadre d'un document CDF. Nous verrons, dans le chapitre 6, l'impact de l'élément `discard` sur la consommation mémoire dans de tels cas. Nous verrons également dans la section 6 comment nous avons exploité cette solution dans une architecture de *streaming* de contenu SVG. Enfin, il faut remarquer que cette solution est simple du point de vue descriptif. Cependant, elle n'est pas très expressive car elle ne permet que le remplacement d'une valeur d'un attribut, qui plus est de la valeur de présentation. Elle n'est pas aussi expressive que les mises à jour Flash, BIFS ou LASeR car elle ne permet pas, entre autres, l'insertion d'éléments. Par ailleurs, elle implique que les mises à jour SVG soient présentes dans le même document que la scène initiale.

2.3.3.3.2 Utilisation d'un langage de programmation

Une autre méthode d'animation des scènes SVG ou HTML est également possible en utilisant la norme DOM (*Document Object Model*). Cette norme définit un modèle et des interfaces pour

manipuler les documents XML. Ce modèle permet à des programmes ou des scripts d'accéder et de modifier de manière dynamique le contenu ou la structure d'un document. L'association d'un document XML (au format SVG, SMIL, XHTML ou autres), du modèle DOM et d'un langage de programmation du type de ECMAScript ou Java donne les moyens, en quelque sorte, de mettre à jour et d'animer le document et la scène qu'il représente. Un exemple déjà très répandu de cette combinaison de technologies est connu sous le nom *« Dynamic HTML »*.

On pourrait donc tout à fait envisager de définir un flux de descriptions de scènes basé sur ces technologies. Pour cela, nous inspirant des exemples précédents, nous définirions une mise à jour comme :

- un ensemble de code ECMAScript;
- suivi d'éventuels fragments de documents XML.

Le code ECMAScript a pour rôle, en faisant appel aux interfaces DOM, de modifier le document. Ces modifications utilisent le cas échéant les données XML fournies dans la mise à jour. Une telle architecture fournit en quelque sorte des mises à jour analogues à celles des formats Flash, BIFS ou LASeR. Les possibilités en termes de mises à jour de ce format sont bien évidemment plus vastes que celles offertes par l'animation discrète SMIL et limitées uniquement par l'interface DOM. Si on ajoute le fait qu'il est naturellement possible de naviguer dans le document XML grâce à DOM et que ce dernier s'intègre aussi bien avec XPath et XPointer, ces mises à jour ne sont pas limitées par un mécanisme d'adressage.

En revanche, en termes de complexité et de vitesse d'exécution, cette solution est plus coûteuse pour plusieurs raisons. Tout d'abord, du point de vue de la taille de l'implémentation et de sa complexité, le surcoût d'une interface DOM est non négligeable dans le développement d'un moteur de composition de scène. Pour cette raison notamment, la version 1.2 du profile *Tiny* de la norme SVG spécifie une version simplifiée de DOM, appelé MicroDOM [55]. Cette version simplifiée dédiée aux documents SVG utilise notamment des objets typés et non des chaînes de caractères ; et réduit les appels possibles. Néanmoins, même dans ce cas, le temps nécessaire pour l'exécution du moteur de script est également important comparé au temps nécessaire à l'exécution d'une mise à jour Flash, BIFS ou LASeR. Le Tableau 2.1 montre le temps moyen nécessaire pour 1000 exécutions de mises à jour LASeR par rapport au temps de 1000 exécutions d'un script MicroDOM effectuant les mêmes opérations dans l'implémentation que nous avons utilisée décrite au chapitre 7.

Mise à jour LASeR	Code MicroDOM/ECMAScript
16,82	604,89

Tableau 2.1 – Durée (en millisecondes) de 1000 exécutions d'un ensemble de mises à jour LASeR par rapport à 1000 exécutions d'un code MicroDOM/ECMAScript équivalent

2.3.3.3.3 REX

Devant l'absence de méthode déclarative normative, complète, efficace et compatible avec les standards DOM et XML pour mettre à jour et animer un document XML, le W3C a décidé en 2006 de démarrer une activité de spécification d'un langage déclaratif pour permettre la modification d'un document XML. Cette nouvelle spécification, encore à l'état d'ébauche, au moment où cette thèse est rédigée, se nomme REX (*Remote Events for XML*) [59]. REX est un langage XML permettant de représenter les évènements de la spécification *DOM 3 Events* [60].

Dans ce langage, les évènements sont décrits en XML par des éléments `event`. Un élément `event` peut décrire plusieurs évènements de même type mais dont les cibles sont différentes. Les cibles sont adressées par une expression XPath. Le principe du langage REX est que ces éléments `event` peuvent être transmis à un terminal distant dans un message REX.

Quand un terminal distant reçoit un message REX, la spécification indique :

- qu'il doit déclencher le ou les évènements décrits par chaque élément `event`;
- et si un des évènements déclenchés est un évènement de modification de l'arbre (*mutation event*), qu'il doit, en plus, appliquer à l'arbre DOM la modification qui aurait pour conséquence de déclencher cet évènement.

Les évènements de modification de l'arbre actuellement envisagés dans la spécification REX sont : `DOMNodeInserted`, `DOMNodeRemoved`, `DOMAttrModified` et `DOMCharacterDataModified`.

La solution REX permet donc, de manière déclarative, d'insérer ou de supprimer des éléments et de modifier le contenu d'un attribut ou le contenu textuel d'un élément dans un arbre DOM. Il s'agit d'une solution équivalente aux mises à jour BIFS ou LASeR à la différence près que la syntaxe REX ne permet pas actuellement de faire des manipulations sur des données typées. Il n'est pas possible par exemple d'exprimer que l'on souhaite remplacer la nième valeur dans un attribut contenant plusieurs valeurs. De même, il n'est pas possible de remplacer une valeur par la somme de cette valeur et de la nouvelle valeur, comme l'autorise le langage LASeR.

Enfin, l'inconvénient de cette solution est le coût dû à la lecture du XML. Ce coût pourrait néanmoins être réduit ou supprimé par des méthodes de compression appropriées comme nous le verrons dans le chapitre 3, mais il faudrait dans ce cas quelques modifications aux modèles DOM ou MicroDOM pour ajouter ces API.

2.3.4 Synthèse sur les méthodes de mise à jour de scènes

Nous avons vu qu'il existait différents langages pour effectuer des mises à jour discrètes d'une scène multimédia, mais que tous suivent la même approche : l'envoi d'instructions, qui doivent être

exécutées soit à la réception, soit à un temps explicite dans la base de temps, soit suite à un évènement utilisateur. La différence entre les langages réside principalement dans le vocabulaire et dans la manière de formuler ces instructions : déclarative ou programmatique, utilisant des évènements, des animations discrètes ou des mises à jour …

2.4 Conclusion

Nous avons présenté les deux grandes approches déclaratives de l'état de l'art pour l'animation de descriptions de scènes que sont : l'interpolation et la mise à jour discrète. L'utilisation d'une approche plutôt que l'autre dépend de deux facteurs : la méthode utilisée pour créer les animations et le scénario de consommation de la scène. Nous avons vu que pour les animations continues, un bon candidat pour représenter ces animations était le mécanisme d'interpolation et que pour les autres cas, les mécanismes de mise à jour étaient plus pertinents.

En ce qui concerne le scénario de consommation de la scène, on distingue deux cas. Dans le cas où la scène est obtenue par téléchargement, par exemple en utilisant un protocole du type HTTP [105], MMS [106] ou FLUTE [76], le lecteur n'ayant plus de contact avec le serveur une fois le téléchargement terminé, il est indispensable que tous les états de la scène soient connus dès le début de la lecture. Une description par interpolation est donc appropriée.

A l'inverse, dans le cas où la scène est obtenue par un protocole de diffusion continue (*streaming broadcasting*, ou téléchargement progressif), la description par trame d'animation offre plus de flexibilité car le serveur peut décider au milieu de la session, sur interaction utilisateur ou sur une décision du diffuseur, de modifier les états suivants de l'animation. Dans ce cas, l'animation par mise à jour est préférable.

Le Tableau 2.2 récapitule les avantages des deux modèles. Bien sûr, les modèles peuvent être combinés dans un même langage. C'est le cas de BIFS qui intègre le modèle d'interpolation repris de VRML mais ajoute le modèle par trame d'animation. C'est également le cas du format LASeR. Il est ainsi possible, par exemple dans un scénario de *streaming*, d'insérer dans la scène ou de modifier, via des mises à jour, des animations décrites par des interpolations. Il faut néanmoins prendre garde que la modification par mise à jour d'une animation en cours d'exécution peut avoir un impact sur la fluidité de l'animation car la mise à jour peut invalider les précalculs faits à l'initialisation de l'animation.

Critère de comparaison	Modèle par trame d'animation	Modèle d'animation par interpolation
Applicabilité	Décrit des animations non paramétriques, non continues. Besoin de mises à jour complexes.	Décrit des animations continues du temps. Connaissance de tous les états de l'animation au début de la diffusion.
Relation avec la notion de temps de scène	Déroulement de l'animation indiqué par le flux (possibilité de pilotage par un serveur).	Contrôle du déroulement de l'animation plus fin au niveau client (choix de la fréquence de rafraîchissement) mais nécessite le maintien d'une notion de temps. Animation en boucle possible.
Relation avec les mécanismes de transport	Possibilités de stockage et transport plus importantes, en utilisant les mécanismes de synchronisation existants.	Compacité de la représentation Ne nécessite pas de voie descendante permanente.
Complexité	Nécessite l'implémentation d'un mécanisme d'application d'une mise à jour.	Nécessite l'implémentation d'un moteur d'animation et d'un mécanisme d'application de l'animation.

Tableau 2.2 – Récapitulatif des avantages des modèles d'animations

Chapitre 3 Descriptions de scènes et Compression

3.1 Introduction

Dans ce chapitre, nous présentons tout d'abord les caractéristiques des descriptions de scènes qui s'avèrent pertinentes pour la compression, en introduisant en particulier la distinction primordiale entre données et structure de scène. Puis nous donnons un état de l'art des techniques de compression de la structure de scène ainsi que certaines techniques de codage de données.

Nous nous sommes intéressés à la compression de descriptions de scènes car il nous est vite apparu que les descriptions de scènes complexes pouvaient être volumineuses. C'est le cas par exemple pour les contenus graphiques vectoriels. Mais ce n'est pas toujours le cas. Quand la scène ne contient pas beaucoup d'objets, quand elle est statique ou de courte durée, la compression n'a pas beaucoup d'intérêts en terme de gain de débit. Il faut également relativiser l'intérêt de ce gain en prenant en compte la taille de la scène, compressée ou non, par rapport à la taille des flux médias associés à la scène. Ainsi, si une scène ne décrit que quelques objets graphiques, quelques interactions possibles et quelques animations, sa taille, compressée ou non, sera négligeable comparée à celle d'un flux audio ou vidéo. Par exemple, dans le cas d'une application de type slideshow d'images, une scène SVG représentant une séquence de 7 images occupe 4,5 Ko à comparer avec les 7 images au format JPEG, de dimension 300x200 qui occupent 55 Ko. Néanmoins, même dans les cas où la compression n'a que peu d'intérêt en débit, la représentation binaire est toujours intéressante car elle permet une lecture plus rapide comme nous le verrons dans le chapitre 7.

Une scène véhicule une information structurée comme nous l'avons décrit dans le chapitre 1. Si l'on s'intéresse à la compression de descriptions de scènes, il faut donc connaître le rapport entre le volume nécessaire pour la représentation de l'information et celui nécessaire pour la représentation de la structure.

Il est intéressant de noter le travail du groupe *Efficient XML Interchange* (EXI) au sein du W3C [61][62]. Ce groupe étudie les mécanismes génériques de compression de document XML. Il propose une classification des contenus XML selon la taille du fichier et la densité d'information. La densité d'information est définie par ce groupe comme le rapport entre le nombre d'octets utilisés pour décrire

le contenu textuel (dans les attributs ou entre les balises XML) et la taille du document en octets. Ainsi, un document est considéré comme très structuré si la densité d'information est inférieure à 50%. A l'inverse, il est considéré comme peu structuré si la densité d'information est supérieure à 50%, c'est-à-dire si le fichier contient plus de données que de balises. Les résultats de ce groupe indiquent 3 catégories de fichier XML : les fichiers peu structurés, les fichiers de petites tailles très structurés, et enfin, les fichiers volumineux très structurés.

Notre expérience est légèrement différente. En effet, avec les documents XML représentant des descriptions de scène, nous n'avons pas rencontré de contenu à la fois très structuré et très volumineux. D'après notre expérience des descriptions de scènes SVG ou XMT, il existe deux types de contenus : les contenus peu volumineux où la part de la structure est prédominante – contenus interactifs du type portail, avec peu de graphiques vectoriels et peu de texte – et les contenus volumineux où la part des données est très importante – contenus du type dessins animés ou du type cartographique. La Figure 3.1 indique le résultat du calcul de densité d'information sur les séquences SVG utilisées dans les phases de conformité du langage LASeR. La taille des fichiers SVG est indiquée en abscisse, en octets, selon une échelle logarithmique. On distingue, sauf pour quelques exceptions, que la densité d'information n'est supérieure à 50% que pour les contenus volumineux.

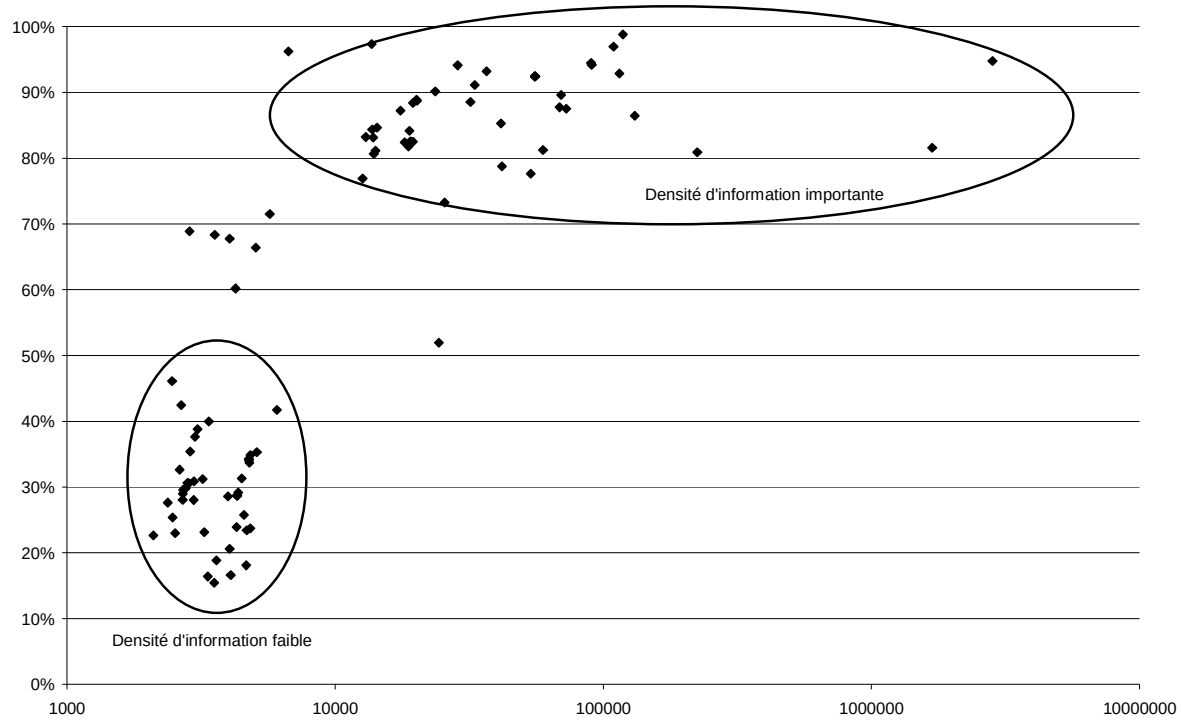


Figure 3.1 – Densité d'information dans les séquences SVG pour la conformité LASeR en fonction de la taille en octets du fichier SVG

A partir de ce constat, la problématique de la compression de descriptions de scènes est donc double. Il s’agit de trouver des mécanismes de compression des informations de structure efficaces en

complexité de décodage ou offrant des caractéristiques de scalabilité pour coder les contenus très structurés. Dans le même temps, il faut trouver des mécanismes de compression de données efficaces, avec ou sans perte, pour réduire significativement le débit des contenus, peu structurés, les plus volumineux, tout en maîtrisant le rapport facteur de compression/qualité. Enfin, il faut combiner les techniques pour les documents très structurés et les documents transportant beaucoup de données sans en perdre les avantages respectifs.

3.2 Compression de la structure des scènes multimédia

Nous présentons dans cette section les méthodes utilisées pour compresser la structure des descriptions de scènes. La compression dans ce domaine consiste à représenter sous forme binaire la structure des descriptions de scènes, c'est-à-dire l'arbre de scène, sans perte, mais en réduisant la redondance de cette structure. On parle également de « binarisation ». Il est intéressant de noter que certaines techniques de réduction de la structure impliquent également une réduction des données à coder. Par exemple dans le domaine XML, moins il y a d'éléments à coder, moins il y a d'attributs à coder. Parmi ces techniques, nous présentons les techniques de prétraitement qui permettent de réduire l'information redondante avant la « binarisation ». Nous présentons également un rapide état de l'art des techniques de représentation binaire de documents structurés.

3.2.1 Factorisation des structures redondantes

L'efficacité de la compression de la structure des descriptions de scènes est fortement dépendante de la représentation non compressée choisie et de la connaissance de la redondance dans cette représentation. En effet, il est possible à partir d'une même scène issue d'un même langage de descriptions d'aboutir à des représentations différentes plus ou moins redondantes. Les exemples ci-dessous illustrent comment un arbre de scène VRML (Figure 3.2) peut être représenté différemment en XMT-A (Code 3.2), XMT-O (Code 3.3) et X3D (Code 3.1). Les exemples de codes XML omettent, pour plus de concision, les déclarations d'espaces de noms, l'instruction indiquant l'encodage du fichier et d'autres informations non nécessaires pour la compréhension.

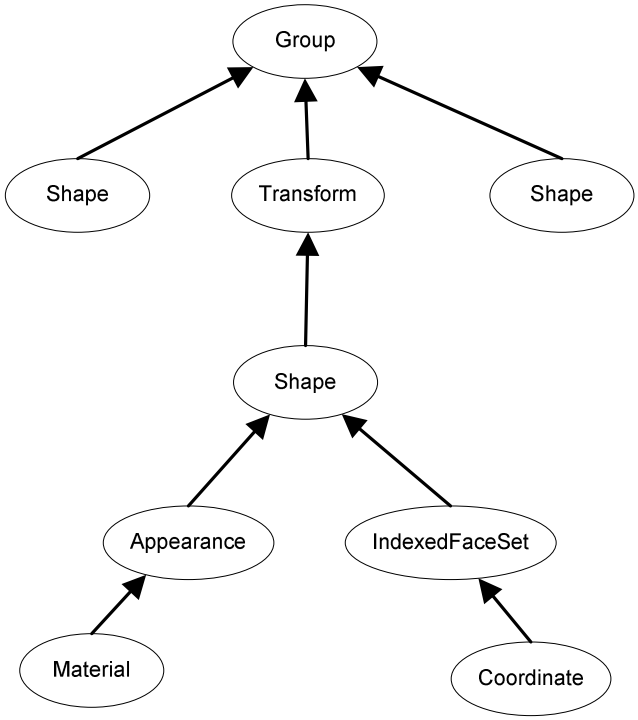


Figure 3.2 – Exemple d'arbre de scène VRML

```
<X3D>
 <Scene>
  <Group>
   <Shape>...</Shape>
   <Transform>
    <Shape>
     <Appearance>
      <Material diffuseColor="1 0 0"/>
     </Appearance>
     <IndexedFaceSet>
      <Coordinate point="0 0 0 0.2 0.2 0.2 0.4 0 0 0.6 0.2 0.2 0.8 0 0 1
  0.2 0.2"/>
     </IndexedFaceSet>
    </Shape>
   </Transform>
   <Shape>...</Shape>
  </Group>
 </Scene>
</X3D>
```

Code 3.1 – Représentation au format X3D de la Figure 3.2

```
<XMT-A>
 <Body>
  <Replace>
   <Scene>
    <Group>
     <children>
      <Shape>...</Shape>
      <Transform>
       <children>
        <Shape>
```

```
          <appearance>
           <Appearance>
            <material>
             <Material diffuseColor="1 0 0"/>
            </material>
           </Appearance>
          </appearance>
          <geometry>
           <IndexedFaceSet>
            <coord>
             <Coordinate point="0 0 0 0.2 0.2 0.2 0.4 0 0 0.6 0.2 0.2 0.8 0
  0 1 0.2 0.2"/>
            </coord>
           </IndexedFaceSet>
          </geometry>
         </Shape>
        </children>
       </Transform>
       <Shape>...</Shape>
      </children>
     </Group>
    </Scene>
   </Replace>
  </Body>
</XMT-A>
```

Code 3.2 – Représentation au format XMT-A de la Figure 3.2

```
<XMT-O>
 <body>
  <par>
   <group>
    <mesh>...</mesh>
    <transformation>
     <mesh coord="0 0 0; 0.2 0.2 0.2; 0.4 0 0; 0.6 0.2 0.2; 0.8 0 0; 1 0.2
  0.2">
      <material diffuseColor="1 0 0"/>
     </mesh>
    </transformation>
    <mesh>...</mesh>
   </group>
  </par>
 </body>
</XMT-O>
```

Code 3.3 – Représentation au format XMT-O de la Figure 3.2

Dans ces exemples, la représentation au format XMT-A diffère de la représentation au format X3D car elle utilise des éléments XML supplémentaires (`children`, `appearance`). Cependant, ces derniers ne sont pas strictement nécessaires. En effet, le nœud `Appearance` ne peut être présent, selon le langage VRML, que dans la propriété `appearance`. De même, la représentation au format XMT-O va encore plus loin dans la démarche et supprime des éléments non primordiaux, qui peuvent être déduits de la connaissance du langage VRML. Cette représentation cherche à regrouper le maximum d'attributs par élément et de minimiser le nombre d'éléments XML. C'est également le choix fait dans le langage SVG comme nous le verrons dans le chapitre 7.

Cependant, d'un point de vue binaire, même si la représentation XMT-A est plus verbeuse que la représentation XMT-O, le train binaire BIFS représentant cette scène sera le même. Cet exemple illustre le fait qu'il est important pour la compression d'une scène décrite en XML, de connaître parfaitement le format XML choisi pour représenter cette scène, notamment ses aspects redondants, afin de les exploiter au maximum lors de la phase de compression.

Il existe des méthodes de codage binaire de documents XML qui utilisent ce principe en exploitant l'information donnée dans la grammaire décrivant le langage XML. Une de ces méthodes est utilisée par le format BiM [47] qui se base sur une analyse du Schema XML [63]. D'autres techniques permettent de réduire la taille du document XML en supprimant de la redondance au niveau de la structure. Ces techniques se présentent sous la forme de prétraitements appliqués avant binarisation. Nous en détaillons certaines dans la suite de ce chapitre.

3.2.1.1 Réutilisation de sous-arbres

Une première technique s'applique aux documents XML dans lesquels des sous-arbres sont entièrement identiques (arborescence d'éléments, attributs spécifiés, valeurs des attributs, et contenus textuels). Dans ce cas, certains langages offrent la possibilité de réutiliser un sous-arbre déjà décrit. On peut citer notamment SVG et XMT. Ces langages utilisent respectivement un élément `use` (pointant vers un élément avec un attribut `id`) et un attribut `use` (pointant vers un élément avec un attribut `DEF`) comme décrit dans les exemples ci-dessous (Code 3.4 et Code 3.5). Dans les deux cas, on attribue un identifiant au sous-arbre à réutiliser et le sous-arbre redondant est remplacé par un élément indiquant l'identifiant du sous-arbre à répliquer. Cette technique est relativement simple à utiliser quand l'information de redondance est disponible dès la création du contenu. Cela peut être exploité dans un outil auteur lorsque l'utilisateur utilise des fonctionnalités de copier/coller. Mais dans le cas contraire, l'exploitation de cette technique nécessite une analyse complexe de l'arbre de scène pour détecter des sous-arbres identiques.

La limitation principale de cette technique est qu'elle ne peut plus être utilisée dès qu'un seul attribut d'un seul élément est différent d'un sous-arbre à l'autre. Cela implique notamment qu'on ne peut pas animer un sous arbre indépendamment de l'autre.

```
<OrderedGroup>
 <children>
  <Shape>
   <appearance>
    <Appearance>
     <material>
      <Material2D emissiveColor="1 0 0" filled="TRUE">
       <lineProps>
        <LineProperties lineColor="0 1 0"/>
       </lineProps>
      </Material2D>
     </material>
```

```
    </Appearance>
   </appearance>
   <geometry>
    <IndexedFaceSet2D>
     <coord>
      <Coordinate2D point="0 0 100 100 150 20 200 0"/>
     </coord>
    </IndexedFaceSet2D>
   </geometry>
  </Shape>
  <Transform2D>
   <children>
    <Shape>
     <appearance>
      <Appearance>
       <material>
        <Material2D emissiveColor="1 0 0" filled="TRUE">
         <lineProps>
          <LineProperties lineColor="0 1 0"/>
         </lineProps>
        </Material2D>
       </material>
      </Appearance>
     </appearance>
     <geometry>
      <IndexedFaceSet2D>
       <coord>
        <Coordinate2D point="0 0 100 100 150 20 200 0"/>
       </coord>
      </IndexedFaceSet2D>
     </geometry>
    </Shape>
   </children>
  </Transform2D>
 </children>
</OrderedGroup>
```

Code 3.4 – Exemple de scène XMT-A sans utilisation de l'attribut USE

```
<OrderedGroup>
 <children>
  <Shape DEF="MaShape">
   <appearance>
    <Appearance>
     <material>
      <Material2D emissiveColor="1 0 0" filled="TRUE">
       <lineProps>
        <LineProperties lineColor="0 1 0"/>
       </lineProps>
      </Material2D>
     </material>
    </Appearance>
   </appearance>
   <geometry>
    <IndexedFaceSet2D>
     <coord>
      <Coordinate2D point="0 0 100 100 150 20 200 0"/>
     </coord>
    </IndexedFaceSet2D>
   </geometry>
  </Shape>
```

```
  <Transform2D>
   <children>
    <Shape USE="MaShape"/>
   </children>
  </Transform2D>
 </children>
</OrderedGroup>
```

Code 3.5 – Exemple de scène XMT-A utilisant l'attribut USE

```
<svg >
 <defs>
  <g fill="red" stroke="yellow" stroke-width="3">
   <rect id="usedRect" width="20" height="20"/>
   <circle id="usedCircle" cx="10" cy="10" r="10"/>
   <ellipse id="usedEllipse" cx="10" cy="10" rx="10" ry="10"/>
   <line id="usedLine" x1="0" y1="10" x2="20" y2="10"/>
   <path id="usedPath" d="M 0 0 L 20 0 L 20 20 L 0 20 Z"/>
   <polygon id="usedPolygon" points="0,0 20,0 20,20 0,20 0 0"/>
   <polyline id="usedPolyline" points="0,0 20,0 20,20"/>
   <g id="usedG">
    <rect width="10" height="20"/>
    <rect x="10" width="10" height="20" fill="rgb(0,128,0)"/>
   </g>
   <use id="usedUse" xlink:href="#usedRect"/>
   <image id="usedImage" xlink:href="../images/20x20.png" width="20"
  height="20"/>
   <text id="usedText">Text</text>
  </g>
 </defs>
 <g transform="translate(150, 25)">
  <use xlink:href="#usedRect" fill="#0F0"/>
  <use y="30" xlink:href="#usedCircle" fill="#0F0"/>
  <use y="60" xlink:href="#usedEllipse" fill="#0F0"/>
  <use y="90" xlink:href="#usedLine" stroke="#0F0" stroke-width="2"/>
  <use y="120" xlink:href="#usedPolyline" stroke="#0F0" stroke-width="2"
   fill="none"/>
  <use y="150" xlink:href="#usedPolygon" fill="#0F0"/>
  <use y="180" xlink:href="#usedPath" fill="#0F0"/>
  <use y="210" xlink:href="#usedImage" fill="#FF0"/>
  <use y="260" xlink:href="#usedText" fill="#0F0" font-weight="bold" font-
   size="25" font-style="italic"/>
  <use x="180" y="0" xlink:href="#usedG" fill="#0F0"/>
  <use x="180" y="30" xlink:href="#usedUse" fill="#0c0"/>
 </g>
</svg>
```

Code 3.6 – Exemple de scène SVG utilisant des éléments use (extrait des séquences de conformité SVG)

La différence entre les mécanismes de réutilisation de sous arbre entre les formats XMT et SVG, outre le fait que l'un utilise un attribut et l'autre un élément, est la possibilité en SVG de coupler la réutilisation avec le mécanisme d'héritage défini par la norme CSS qui permet de remédier à la limitation décrite précédemment, à savoir qu'on peut réutiliser tout un sous-arbre avec des valeurs d'attributs différentes car héritées en fonction de la position du sous-arbre réutilisé.

3.2.1.2 Héritage de propriétés

Une seconde technique pour réduire la redondance dans les documents décrivant des descriptions de scènes est une technique issue de la norme CSS. Il s'agit initialement d'une technique qui a pour but d'associer des propriétés à certains éléments d'un même sous arbre dans le but d'avoir une cohérence visuelle, un même style. Cette technique s'appelle l'héritage. L'héritage consiste à spécifier une certaine valeur pour un attribut sur un élément particulier, pour que cette valeur soit transmise de proche en proche dans les descendants de cet élément jusqu'à ce qu'un élément spécifie une autre valeur pour cet attribut. Selon la norme CSS, seuls certains attributs peuvent être hérités, on les appelle des propriétés. Ces propriétés ont une valeur particulière, généralement valeur par défaut, qui indique que la valeur de cet attribut dans l'arbre doit être déduite par héritage. Il s'agit de la valeur *inherit*.

Cette technique au départ prévue pour des aspects visuels peut tout à fait être utilisée pour réduire la quantité d'information à coder dans un document XML, aussi bien des informations de structure que de données puisqu'elle réduit le nombre d'apparitions d'un attribut, et donc de sa valeur, dans l'arbre. L'exemple ci-dessous montre un fragment de la description d'une scène sans héritage (Code 3.7) et avec héritage implicite (Code 3.8), car la valeur des attributs non spécifiés sur certains éléments (`stroke`, `stroke-width`, `fill`) est la valeur par défaut : *inherit*.

```
<g id="S1" >
<path fill="rgb(90,91,93)" stroke="rgb(60,29,8)" stroke-width="2.0"
  d="M307.3 -139.5 ... z" />
<path fill="rgb(154,122,83)" stroke="rgb(60,29,8)" stroke-width="2.0"
  d="M668.0 -187.5 ... z" />
<path fill="rgb(182,200,232)" stroke="rgb(60,29,8)" stroke-width="2.0"
  d="M325.05 -269.0 ... z" />
<path fill="rgb(139,98,74)" stroke="rgb(60,29,8)" stroke-width="2.0"
  d="M602.5 -68.5 ... z" />
<path fill="rgb(139,98,74)" stroke="rgb(60,29,8)" stroke-width="2.0"
  d="M689.65 -53.5 ... z" />
<path fill="rgb(106,78,63)" stroke="rgb(60,29,8)" stroke-width="2.0"
  d="M283.5 -84.5 ... z" />
<path fill="rgb(106,78,63)" stroke="rgb(60,29,8)" stroke-width="2.0"
  d="M395.85 -51.0 ... z" />
<path fill="rgb(177,96,37)" stroke="rgb(60,29,8)" stroke-width="2.0"
  d="M288.0 -90.0 ... z" />
<path fill="rgb(223,223,223)" stroke="rgb(60,29,8)" stroke-width="2.0"
  d="M602.5 -201.0 ... z" />
<path fill="rgb(238,176,116)" stroke="rgb(60,29,8)" stroke-width="2.0"
  d="M640.2 -347.9 ... z" />
<path fill="rgb(238,176,116)" stroke="rgb(60,29,8)" stroke-width="2.0"
  d="M684.5 -219.5 ... z" />
<path fill="rgb(101,101,101)" stroke="rgb(60,29,8)" stroke-width="2.0"
  d="M620.0 -304.0 ... z" />
<path fill="rgb(238,176,116)" stroke="rgb(60,29,8)" stroke-width="2.0"
  d="M593.0 -105.0 ... z" />
<path fill="none" stroke="rgb(60,29,8)" stroke-width="2.0" d="M653.5 -329.5
  ... 668.45 -336.9"/>
</g>
```

Code 3.7 – Exemple de fragment de scène SVG sans héritage

```
<g id="S1"
   fill="rgb(139,98,74)" stroke="rgb(60,29,8)" stroke-width="2.0">
<path fill="rgb(90,91,93)" d="M307.3 -139.5 ... z" />
<path fill="rgb(154,122,83)" d="M668.0 -187.5 ... z" />
<path fill="rgb(182,200,232)"d="M325.05 -269.0 ... z" />
<path d="M602.5 -68.5 ... z" />
<path d="M689.65 -53.5 ... z" />
<path d="M283.5 -84.5 ... z" />
<path fill="rgb(106,78,63)" d="M395.85 -51.0 ... z" />
<path fill="rgb(177,96,37)" d="M288.0 -90.0 ... z" />
<path fill="rgb(223,223,223)" d="M602.5 -201.0 ... z" />
<path fill="rgb(238,176,116)" d="M640.2 -347.9 ... z" />
<path fill="rgb(238,176,116)" d="M684.5 -219.5 ... z" />
<path fill="rgb(101,101,101)" d="M620.0 -304.0 ... z" />
<path fill="rgb(238,176,116)" d="M593.0 -105.0 ... z" />
<path fill="none" d="M653.5 -329.5 ... 668.45 -336.9"/>
</g>
```

Code 3.8 – Exemple de fragment de scène SVG avec héritage implicite

Dans le langage SVG, l'héritage permet d'aller plus loin dans la réduction de la redondance de structure car on peut combiner héritage et utilisation de l'élément `use` pour réutiliser des sous arbres dont les propriétés diffèrent.

L'héritage est un mécanisme intéressant par sa simplicité d'utilisation, il facilite la tâche des créateurs de scènes. On peut également, par un traitement automatisé simple, qui consisterait à remonter l'arbre de scène, détecter les propriétés communes et les factoriser. Cependant, il ne permet de réduire la redondance qu'au niveau de certains attributs. De plus, il peut s'avérer complexe à interpréter au niveau du lecteur de contenu surtout quand il est combiné avec des animations. Cet aspect, déjà abordé dans le chapitre 2, sera décrit plus en détail dans le chapitre 7.

3.2.1.3 Factorisations avancées

Il existe des mécanismes plus avancés pour réduire la redondance dans la structure de la description d'une scène. Ces mécanismes ont été ou sont développés pour faciliter la tâche d'édition, exploiter plus efficacement les copier/coller. En effet, souvent lors de la création d'une scène, l'auteur utilise les fonctions de copier/coller, puis il modifie légèrement la partie collée. Cela rend impossible la réutilisation de sous arbre. L'idée d'une factorisation avancée est d'introduire une notion d'objet, décrivant un sous arbre à réutiliser, avec des paramètres accessibles et modifiables. L'association de cette technique et des techniques de création de scènes hiérarchiques permet la création de bibliothèques d'objets de scènes (comme les *widgets* décrit au chapitre 1) auxquels sont associés des comportements. Nous décrivons ci-après certaines de ces techniques.

3.2.1.3.1 Prototypes VRML

Un *Proto* est une primitive VRML (et BIFS) qui permet de définir des nœuds ad-hoc en se basant sur des nœuds définis dans le standard. Elle s'utilise en deux temps. Dans un premier temps, on déclare la définition du *Proto* en donnant son équivalent dans le standard sous la forme d'une sous-scène

(nœuds, routes et éventuellement *Proto*) et ensuite dans un second temps, on utilise ce nouveau nœud comme n'importe quel autre nœud. Enfin, on peut définir des *Protos* dans une autre scène et les importer dans une nouvelle scène, ce qui permet de définir des bibliothèques de nœuds spécialisés. On parle alors d'*Extern Proto*. Le Code 3.9 décrit la déclaration d'un *Proto*.

```
<ProtoDeclare name="PForm" protoID="0">
  <field name="appearance" type="Node" vrml97Hint="field" />
  <field name="point" type="Vector2Array" vrml97Hint="exposedField"
   vector2ArrayValue="" />
  <Shape>
    <geometry>
      <IndexedFaceSet2D colorPerVertex="false">
       <coord>
         <Coordinate2D>
           <IS><connect nodeField="point" protoField="point"/></IS>
         </Coordinate2D>
       </coord>
     </IndexedFaceSet2D>
    </geometry>
    <IS><connect nodeField="appearance" protoField="appearance"/></IS>
  </Shape>
</ProtoDeclare>
```

Code 3.9 – Exemple de déclaration en XMT d'une primitive *Proto*

Dans cet exemple, la balise `ProtoDeclare` déclare un *Proto* dont le nom est `PForm`, qui possède une interface avec deux propriétés `field`, appelées respectivement `appearance` et `point`, pouvant respectivement contenir un nœud (`type="Node"`) et une liste de points (`type="Vector2Array"`). On donne la définition, le corps de la macro, sous forme de nœuds BIFS standards de ce *Proto*, à savoir un nœud `Shape` qui contient un nœud `IndexedFaceSet2D` … On indique également que : le champ `appearance` du *Proto* est connecté (`connect`) au champ `appearance` du nœud `Shape` de sa définition; et de même, que le champ `point` du *Proto* correspond au champ `point` du nœud `Coordinate2D`. L'utilisation de ce *Proto* est décrite dans le Code 3.10.

```
<ProtoInstance name="PForm">
  <fieldValue name="appearance">
    <node><Appearance USE="ap1"/></node>
  </fieldValue>
  <fieldValue name="point" vector2ArrayValue="-103.0 -290.0 …"/>
</ProtoInstance>
```

Code 3.10 – Exemple d'utilisation en XMT d'une primitive *Proto*

On indique donc la présence d'une instance de *Proto* par la balise `ProtoInstance`, et uniquement la valeur des champs de ce nouveau nœud. Cette instance se comporte de la même manière que les autres nœuds par rapport aux évènements de la scène principale, c'est-à-dire qu'on peut lui donner un identifiant et l'utiliser dans des routes pour véhiculer des évènements entrant ou sortant de cette instance. Elle possède également un comportement intrinsèque décrit dans le corps de la déclaration,

par des routes, des capteurs d'évènements, des nœuds `TimeSensor`, tout comme dans une scène complète.

La structure interne du *Proto* pourrait être très complexe et volumineuse, la manière de l'utiliser serait la même. Cela donne une grande flexibilité pour créer des scènes si on y pense dès la création. Mais, à nouveau, si l'auteur n'indique pas qu'il souhaite créer un *Proto*, il est difficile de manière automatique de détecter ou de trouver le ou les *Protos* qui réduiront la structure de la scène de manière optimale. Nous donnerons un exemple de ce problème dans le chapitre 6.

3.2.1.3.2 Les technologies W3C : XSLT, XBL et sXBL

Le W3C a vu également apparaître quelques techniques qui indirectement servent à réduire la redondance des scènes. Nous avons déjà vu le mécanisme d'héritage issu de la norme CSS. Il existe également d'autres techniques permettant d'associer des styles aux documents XML ou plus généralement de transformer un document XML A en un autre document XML B, B étant une représentation plus compacte de A.

3.2.1.3.2.1 XSL et XSLT

Le W3C a défini les normes XSL « XML Style Sheet » et XSLT « XSL Transformation » [64] qui permettent de spécifier des transformations de documents XML. Ces transformations sont définies par des règles. Chaque règle associe à un ensemble d'éléments et/ou d'attributs *E*, un processus de génération de fragments XML utilisant *E* pour produire un nouvel ensemble d'éléments et/ou d'attributs *F*. *E* ici serait l'équivalent de l'instance d'un *Proto* et F la déclaration, le corps du *Proto*. Les processus de génération de fragments XML peuvent par exemple recopier un ou plusieurs éléments et/ou attributs, créer de nouveaux éléments et/ou attributs, accéder à n'importe quelle partie du document d'origine en utilisant la norme XPath. L'exemple Code 3.11 montre un document XML avant transformation, une transformation (Code 3.12) et le résultat après transformation (Code 3.13). Ces exemples sont issus de la norme XSLT.

```
<card type="simple">
   <name>John Doe</name>
   <title>CEO, Widget Inc.</title>
   <email>john.doe@widget.com</email>
   <phone>(202) 456-1414</phone>
 </card>
```

Code 3.11 – Document XML avant transformation

```
<xsl:stylesheet>
 <xsl:template match="card[@type='simple']">
  <html xmlns="http://www.w3.org/1999/xhtml">
   <title>business card</title>
   <body>
    <xsl:apply-templates select="name"/>
    <xsl:apply-templates select="title"/>
```

```
    <xsl:apply-templates select="email"/>
    <xsl:apply-templates select="phone"/>
   </body>
  </html>
 </xsl:template>

 <xsl:template match="card/name">
  <h1><xsl:value-of select="text()"/></h1>
 </xsl:template>

 <xsl:template match="email">
  <p>email: <a href="mailto:{text()}"><tt>
     <xsl:value-of select="text()"/></tt></a></p>
 </xsl:template>
...
</xsl:stylesheet>
```

Code 3.12 – Règles de transformation au format XSL

```
<html>
 <title>business card</title>
 <body>
   <h1>John Doe</h1>
   <h3><i>CEO, Widget Inc.</i></h3>
   <p>email: <a href="mailto:john.doe@widget.com">
      <tt>john.doe@widget.com</tt></a>
   </p>
   <p>phone: (202) 456-1414</p>
 </body>
</html>
```

Code 3.13 – Document XML après transformation

Ce langage de transformation est très expressif. Il permet, grâce d'une part au mécanisme d'association entre extraits du document et règles de transformation, et d'autre part à la récursivité et à la possibilité d'utiliser des extensions Java ou Javascript, de modifier le document XML de manière illimitée. Cependant, le traitement de la transformation est un processus coûteux qui nécessite souvent d'avoir tout le document en mémoire. Il est difficile, sauf en restreignant l'expressivité du langage XPath, d'exécuter la transformation en temps réel, au moment de la présentation de la scène. Pour cela des alternatives existent comme STX [78] et des études ont été réalisées pour mesurer le gain apporté par des approches de transformations de documents XML au fil de l'arrivée d'évènements SAX (c.f. DANAE [81]). De plus, la transformation XSL n'indique pas comment un fragment de document XML résultant de la transformation s'intègre dans le document final au niveau animation et évènement. Ce sont ces limitations que XBL et sXBL tentent de résoudre.

3.2.1.3.2.2 XBL et sXBL[6]

XBL [65], « eXtensible Binding Language », est un langage XML, défini initialement par le groupe Mozilla [79] et repris par le W3C, qui permet de décrire des associations entre des éléments de différents langages XML. sXBL [66], « SVG's XML Binding Language », en cours de définition au sein du groupe SVG au W3C, spécifie un langage d'association entre un document XML quelconque et un document SVG. Ce langage permet de décrire comment une balise, d'un langage XML quelconque, qui sémantiquement correspond à un objet graphique, vectoriel, avec un comportement, de l'animation, peut-être implémentée en SVG. Ainsi, tout comme le mécanisme VRML de *Proto*, sXBL permet de définir des éléments, non spécifiés par la norme SVG, qu'un lecteur de contenu SVG pourra représenter sous forme d'un arbre SVG en interprétant les balises sXBL au moment de la présentation du document. L'exemple Code 3.14 montre la déclaration d'un élément dont le nom est `HelloWorld` dans l'espace de noms dont le préfixe est `myNS`. L'élément `template` indique que ce nouvel élément correspond à un élément SVG `text`.

```
<xbl:xbl>
 <xbl:definition element="myNS:HelloWorld>
  <xbl:template>
   <text><xbl:content/></text>
  </xbl:template>
 </xbl:definition>
</xbl:xbl>
```

Code 3.14 – Déclaration d'un nouvel élément dans le langage sXBL

L'utilisation de ce nouvel élément se fait de manière très simple, comme décrit dans Code 3.15.

```
<svg>
 <rect x="1" y="1" width="198" height="58" fill="none" stroke="blue"/>
 <g font-size="14" font-family="Verdana" transform="translate(10,35)">
  <myNS:HelloWorld>Hello, world, using sXBL</myNS:HelloWorld>
 </g>
</svg>
```

Code 3.15 – Fragment de document SVG décrivant l'utilisation d'un élément défini en sXBL

[6] Au moment où ce document est rédigé, il semblerait que ces deux langages convergent en un seul langage intitulé XBL2 [64].

Un comportement peut-être associé à chaque élément sXBL en ajoutant dans la déclaration de cet élément des gestionnaires d'évènements (`handler`). De même, on peut définir des éléments sXBL dont le contenu sera dépendant de l'utilisation de ce nouvel élément.

3.2.2 Mécanismes génériques de représentation binaire de documents XML

La compression de la structure des descriptions de scènes implique la réduction des structures redondantes comme nous venons de le voir. Une fois cette étape effectuée, il s'agit de représenter la structure restante sous forme binaire. La plupart des langages passent pour cela par une étape de représentation au format XML puis par une étape de compression générique de document XML. Nous faisons ici une synthèse de l'état de l'art détaillé sur la compression générique de document XML à partir de la thèse de C. Seyrat [29].

Les principales caractéristiques des langages XML expliquent pourquoi les documents XML sont souvent volumineux et inefficaces en compression. En effet, XML est un format texte dont un des buts est la lisibilité. L'utilisation d'un format texte impose l'utilisation de codage de caractères (ASCII, UTF-8, UTF-16) qui ont des propriétés facilitant la recherche, l'édition mais qui ne sont pas efficaces en compression, car à longueur fixe. De plus, les éléments de syntaxe des langages XML, c'est-à-dire les noms des balises (des éléments) et les noms des attributs sont souvent longs pour être compréhensibles par le lecteur. Enfin, les choix faits pour comprendre la structuration du document en le lisant, à savoir la répétition du nom d'une balise quand on ferme cette balise, ainsi que l'utilisation de caractères d'échappement (<,>,< ?, < !, < !--, [[) font des documents XML des fichiers très volumineux.

La représentation binaire des documents XML et l'exploitation de la redondance dans un fichier XML n'est pas difficile à priori. En effet, les algorithmes du type ZLIB [30] obtiennent de bons résultats. Mais la sélection du bon outil de suppression de la redondance, en fonction des propriétés de codage choisies, est un problème plus ténu : certains outils de compressions de données XML peuvent permettre le décodage en mode *streaming* et d'autres non; certains ont besoin de la connaissance de la grammaire correspondant à un document XML pour encoder et décoder le flux alors que d'autres non; certains vont permettre de coder toutes les instructions possibles dans un langage XML (processing instructions, commentaires …) et d'autres non. Le lecteur pourra consulter la liste des propriétés intéressantes concernant le codage de document XML définie par le groupe EXI [62].

Il faut retenir que les codages exploitant une grammaire associée au document permettent de coder plus efficacement un document. Il n'est pas rare d'obtenir un facteur de compression de l'ordre de 20 pour la compression d'un document XML. Mais il faut retenir également que les meilleurs taux de

compression s'obtiennent souvent au détriment de propriétés de temps d'accès, de robustesse aux erreurs, etc.

3.2.3 Exemple : le codage des commandes BIFS

Parmi les langages de descriptions de scènes, il est intéressant d'étudier un peu plus en détail un mécanisme non générique de compression de document XML, celui permettant de représenter sous forme binaire une scène BIFS et plus généralement un flux BIFS.

Un flux BIFS est un flux binaire composé d'unité d'accès (AU) BIFS. Il existe deux types d'AU BIFS : *BIFS-Update* ou *BIFS-Anim*. Nous ne traitons ici que des *BIFS-Updates*. Une AU BIFS de type *BIFS-Update* est composée de commandes BIFS. Les différentes commandes BIFS sont décrites dans la Figure 3.3.

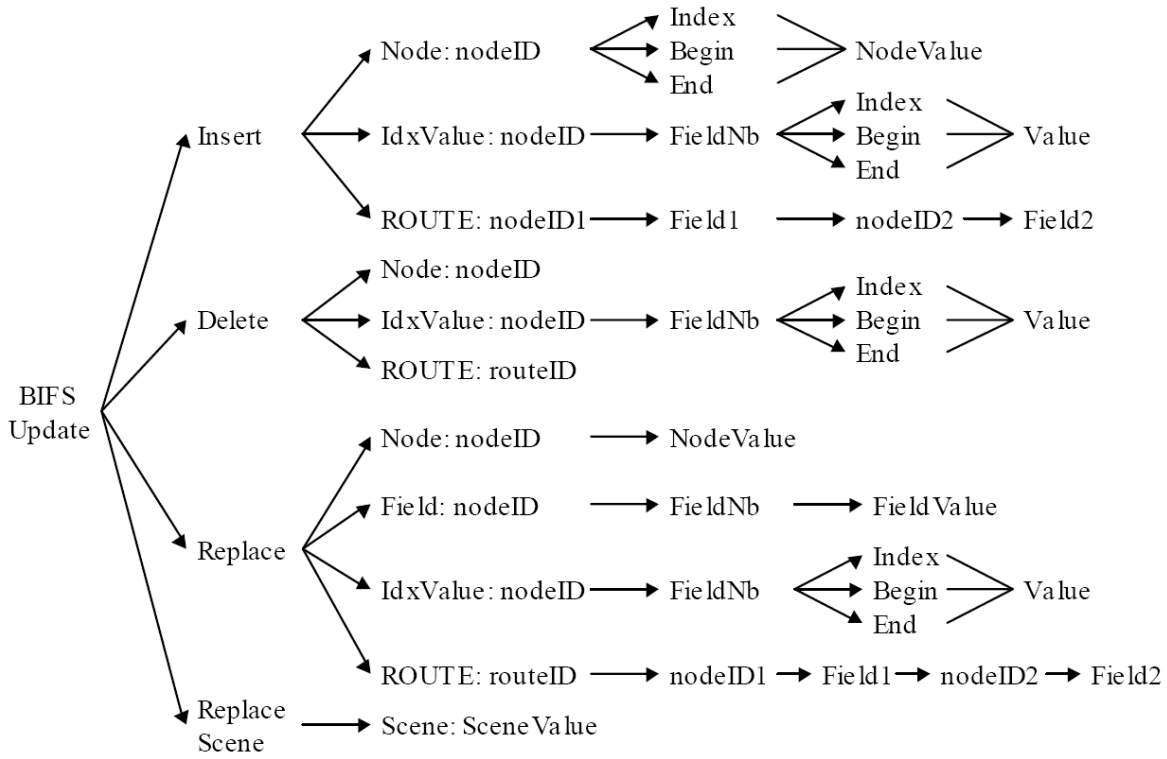


Figure 3.3 – Arborescence simplifiée des commandes BIFS

Une commande BIFS est codée par un type et des données éventuelles. Ces données dépendent du type de commande mais il s'agit généralement soit d'un sous-arbre BIFS soit d'une valeur primitive.

3.2.3.1 Codage des arbres BIFS

Un arbre BIFS est composé de nœuds. Chaque nœud possède des champs. Chaque champ peut contenir une seule valeur primitive, on parle alors de champ de type `SFField` ou plusieurs valeurs, on parle de champ de type `MFField`. Les valeurs primitives peuvent être de type simple, par exemple

un entier, un nombre décimal, un booléen ou une chaîne de caractères ; ou complexe, par exemple de type nœud.

Un arbre BIFS est composé d'un nœud racine possédant des champs qui peuvent contenir des nœuds. Ces nœuds à leur tour peuvent contenir d'autres nœuds dans leurs champs le permettant. Le codage d'un tel arbre est contextuel. Un nœud sera codé différemment s'il est contenu dans un champ ou dans un autre. Le code Code 3.16 décrit le codage d'un nœud en fonction du type de champ dans lequel il se trouve (`nodeDataType`).

```
bit(1) isReused;
if (isReused) {
  bit(nodeIDbits) nodeID;
} else {
  int nodeGroup = 0;
  do {
    nodeGroup++;
    bit(GetNDTnbBits(nodeGroup, nodeDataType)) localNodeType;
  } while (localNodeType == 0);
  nodeType = GetNodeType(nodeGroup, nodeDataType, localNodeType);
}
bit(1) isUpdateable;
if (isUpdateable) {
  bit(nodeIDbits) nodeID;
}
bit(1) MaskAccess;
if (MaskAccess) {
  MaskNodeDescription;
} else {
  ListNodeDescription;
}
```

Code 3.16 – Codage BIFS d'un nœud

Un nœud est donc codé soit uniquement par un identifiant (`nodeID`) s'il a déjà été transmis auparavant (codage de la primitive `use` décrite précédemment), soit par un type (`localNodeType`), sur un nombre variable de bits. Ce type est suivi d'un bit indiquant si le nœud déclare un identifiant (`nodeID`), suivi d'un booléen indiquant comment sont encodés les champs de ce nœud et terminé par le codage des champs.

Le mot de code indiquant le type d'un nœud (`localNodeType`) est contextuel. Si deux nœuds de même type (`nodeType`) sont codés comme appartenant à deux champs de types différents (`nodeDataType`), l'identifiant du type de nœud sera codé différemment (`localNodeType`). Cette particularité du codage BIFS implique l'utilisation de tables de codage.

3.2.3.2 Codage des champs BIFS

En BIFS, chaque champ possède une valeur par défaut. On ne code que les champs qui ont une valeur différente de la valeur par défaut. La liste de ces champs peut être codée de deux manières différentes en fonction de la variable `MaskAccess` : soit sous forme d'une liste chaînée

(`ListNodeDescription`), soit sous forme d'un tableau (`MaskNodeDescription`). Pour une liste chaînée (Code 3.18), on indique pour chaque champ un identifiant du champ dans le nœud courant, suivi du codage de la valeur du champ, suivi d'un bit indiquant si la liste est terminée. Pour le codage sous forme de tableau (Code 3.19), on passe en revue dans un ordre bien précis tous les champs possibles dans le nœud et pour chaque champ, on indique si une valeur est codée ou non. La taille du tableau n'est pas codée car la liste de tous les champs possibles par type de nœud est supposée connue. A nouveau, cela suppose qu'un décodeur BIFS ait en mémoire, par nœud, la liste des champs possibles et un identifiant par champ.

```
for (i=0; i<node.numDEFfields; i++) {
  bit(1) Mask;
  if (Mask) { FieldValue; }
}
```

Code 3.17 – Codage BIFS des champs d'un nœud

```
bit(1) endFlag;
while (!EndFlag){
  bit(node.nDEFbits) fieldRef;
  FieldValue;
  bit(1) endFlag;
}
```

Code 3.18 – Codage BIFS des champs par liste chaînée

3.2.3.3 Codage des listes de valeurs

Pour les champs de type `MFField`, c'est-à-dire qui peuvent contenir plusieurs valeurs primitives, il existe également deux types de codage utilisant de même que précédemment un liste chaîne ou un tableau dont la dimension doit être codée dans le flux car on ne connaît pas à l'avance le nombre de valeurs dans la liste. Le Code 3.19 décrit ce codage.

```
bit(1) isListDescription;
if (isListDescription) {
  bit(1) endFlag;
  while (!endFlag) {
    SFFieldValue;
    bit(1) endFlag;
  }
} else {
  int(5) NbBits;
  int(NbBits) numberOfFields;
  SFFieldValue values[numberOfFields];
}
```

Code 3.19 – Codage BIFS des valeurs dans une liste de valeurs

Les valeurs simples (`SFFieldValue`) de type entier, nombre décimal et booléen sont codées par défaut respectivement sur 32 bits, 32 bits and 1 bit et sur un nombre variable de bits si l'outil de

quantification, décrit plus loin, est activé. Les valeurs simples de type chaîne de caractères sont codées sur 8 bits par caractères et terminées par le caractère 0.

3.3 Compression des données des scènes multimédia

Comme nous l'avons vu en introduction à ce chapitre, dans les descriptions de scènes complexes et volumineuses, une grande partie de l'information à coder est constituée par les données de la scène. Ces données sont généralement des matrices de transformation, des épaisseurs de traits, des coordonnées de points, des triplets de couleurs ou des chaînes de caractères. De manière générale, les valeurs numériques sont représentées par des valeurs décimales pour offrir une précision maximum. Nous présentons ici comment les différents formats de descriptions de scènes binaires représentent ces données de manière compacte. La plupart des langages offrent à la fois des mécanismes de représentations sans perte et avec perte. Nous présentons ces deux aspects dans la suite de cette partie. Nous nous attarderons également sur le codage des données de type objets graphiques qui sont très utilisés dans les scènes multimédia et qui nécessitent une analyse supplémentaire afin d'obtenir un codage efficace.

3.3.1 Mécanismes de représentations binaires sans perte

Nous présentons ici les mécanismes de représentations binaires qui permettent de représenter certaines données de façon efficace, binaire et sans perte.

3.3.1.1.1 Mécanismes Flash

Les données encodées dans un contenu Flash utilisent un codage spécifique à chaque type de données, et ces codages spécifiques sont en nombre limité. Les nombres entiers sont représentés sur 8, 16 ou 32 bits. Les nombres décimaux sont représentés sur 32 bits avec une représentation à virgule fixe 16.16. Mais, dans certains cas, entiers ou nombre décimaux peuvent être codés sans perte sur un nombre variable de bits, dont la longueur est indiquée sur 5 bits. Enfin, en plus de ces mécanismes, le format Flash autorise la compression du fichier final au format ZLIB. Le fichier complet, et non pas chaque trame, est encodé en un seul bloc par le compresseur ZLIB.

Le format Flash est un format binaire compact qui doit sa compacité à certaines de ces particularités du codage. Cependant, le gain principal en compression de la représentation binaire Flash est induit par un mécanisme de quantification des données (avec perte) que nous abordons plus loin dans ce chapitre.

3.3.1.1.2 Mécanismes BIFS

La norme BIFS a défini certains outils pour permettre de coder efficacement les données transportées dans les scènes BIFS. Certains outils permettent un codage sans perte, et d'autres, comme la

quantification linéaire, sont des outils de codage avec perte. Nous présentons les premiers dans cette section et les seconds plus loin dans ce chapitre.

3.3.1.1.2.1 Signalisation

Toutes les techniques que nous présentons ici ne sont pas des techniques utilisées par défaut dans une scène BIFS. Il faut pour les utiliser, le signaler à un encodeur/décodeur BIFS. Cette signalisation se fait par l'utilisation d'un nœud BIFS particulier : le nœud `QuantizationParameter`. Ce nœud signale quelle technique de codage est utilisée pour tous les sous-arbres, dont la racine est située au même niveau dans l'arbre de scène que le nœud `QuantizationParameter`. La Figure 3.4 illustre le champ d'application d'un nœud `QuantizationParameter`. Ce nœud est inséré dans la scène au moment du codage. Au moment du décodage, il est décodé comme tout autre nœud mais comme il n'est pas utilisé dans la phase de rendu, il n'est pas ajouté à l'arbre de scène.

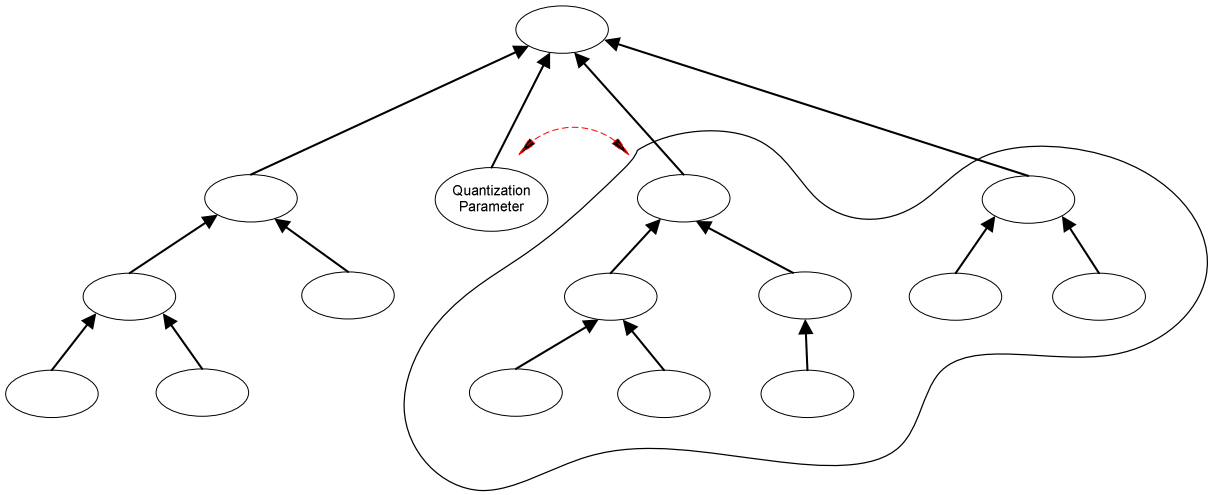


Figure 3.4 – Champ d'application du nœud QuantizationParameter dans une scène BIFS

Ce nœud est coûteux à signaler par rapport aux autres nœuds car il possède beaucoup de propriétés (40). Son utilisation doit donc être le résultat d'une étude du coût de la signalisation par rapport au gain de codage.

3.3.1.1.2.2 Codage des nombres décimaux

L'encodage par défaut des nombres décimaux en BIFS utilise une représentation sur 32 bits définie par la norme IEEE 754. Cette représentation présente l'avantage d'utiliser un nombre fixe de bits mais l'inconvénient d'être inefficace en compression pour des nombres dont la précision est inférieure à 32 bits. Par exemple, il faut 32 bits pour coder la valeur décimale 0.0.

La norme BIFS prévoit un autre codage sans perte et à longueur variable, séparant le codage de l'exposant et de la mantisse, chacun codé en base 2. Ce codage est appelé « *Efficient Float Coding* ». La formule de reconstitution d'un nombre *x* codé selon ce principe est donnée ci-dessous :

$$x = (1 - 2 \cdot mSign) \cdot (2^{mLength-1} + m) \cdot 2^{(1-2 \cdot eSign) \cdot (2^{eLength-1} + e)}$$

Équation 3.1 - Reconstitution d'une valeur décimale *x* codée selon le procédé « Efficient Float Coding »

Dans cette formule, *mSign*, *mLength*, *m*, *eSign*, *eLength* et *e* sont respectivement codés sur *1*, *4*, *mLength–1*, *1*, *3* et *eLength–1* bits.

L'utilisation de ce mécanisme est signalée par l'attribut `useEfficientFloat` du nœud `QuantizationParameter`.

3.3.2 Mécanismes de représentations binaires avec perte

Certains formats binaires de représentation de scènes et de graphiques vectoriels offrent des mécanismes de quantification qui permettent de réduire le volume des données numériques de manière importante. Certains formats utilisent des outils de quantification permettant de choisir la résolution souhaitée pour coder des objets graphiques, c'est le cas de BIFS. D'autres imposent une résolution fixe pour toute la scène, c'est le cas de Flash, il faut dans ce cas créer les objets graphiques avec la bonne précision étant donnée cette résolution. Nous présentons ici ces différentes méthodes de quantification pour proposer ensuite un comparatif.

3.3.2.1 Quantification Flash

La particularité la plus notable du codage Flash est que toutes les coordonnées sont exprimées en *twips*. Le *twip* correspond à 1/20ème de pixel. Tous les points d'un contenu Flash sont donc échantillonnés selon une grille dont la résolution est le 20ème de pixel. Un contenu Flash ne peut pas représenter de données plus précises. Si le facteur de zoom sur un objet devient, lors de la présentation de la scène, supérieur à 20, des artefacts de codage apparaissent. Pour remédier à ce problème, une technique consiste à appliquer, à tous les points de l'objet, le zoom maximal qui sera appliqué dans la scène, et à coder cette nouvelle série de points. Ensuite, cette série de points est placée dans la scène sous la forme d'un objet dont le facteur de zoom varie entre 1 et le facteur inverse à celui utilisé pendant le codage.

3.3.2.2 Quantification BIFS

La norme BIFS offre également la possibilité de coder avec perte les nombres décimaux en effectuant une quantification linéaire. On peut activer ou désactiver la quantification par type de données (couleur, point 2D). C'est le nœud `QuantizationParameter` qui indique, pour chaque type de donnée, si la quantification est activée ou non, et le cas échéant les paramètres de la quantification : borne inférieure v_{min}, borne supérieure v_{max}, nombre de bits *N* pour coder chaque valeur. La formule de codage d'une valeur *v* est la suivante :

$$v_q = \mathrm{int}\left(\frac{v - v_{\min}}{v_{\max} - v_{\min}} \left(2^N - 1\right) \right)$$

La valeur codée est v_q et la fonction *int*() est une fonction qui transforme un nombre décimal en entier. Cette transformation peut être un arrondi ou une troncature, le type de fonction est laissé au libre choix de l'encodeur.

La valeur *v* étant comprise entre v_{min} et v_{max}, le quotient $\frac{v - v_{\min}}{v_{\max} - v_{\min}}$ est compris entre 0 et 1, donc le paramètre de la fonction *int*() est un nombre décimal compris entre 0 et $2^N - 1$. La valeur v_q est donc un entier entre 0 et $2^N - 1$.

Le décodage s'effectue suivant la formule :

$$\hat{v} = v_{\min} + v_q \frac{v_{\max} - v_{\min}}{2^{\max(N,1)} - 1}.$$

On peut déterminer le bon nombre de bits pour coder en fonction de la résolution. En effet, si on définit la résolution *r* comme l'inverse du quotient $\frac{v_{\max} - v_{\min}}{2^N - 1}$, et que l'on inverse la relation liant la résolution *r* et le nombre de bits *N* nécessaires pour coder des points avec cette résolution, on obtient le nombre bits minimum nécessaires pour représenter une résolution précise :

$$N = \log_2\left(1 + \frac{v_{\max} - v_{\min}}{1 / r} \right)$$

Ce procédé de codage produit des artefacts de codage si le nombre de bits *N* choisi pour coder v_q est inférieur au nombre de bits nécessaire pour coder la résolution *r*. Pour illustrer cette relation, nous quantifions un ensemble de points, formé de 128x128 points, dont les coordonnées sont les entiers compris entre 0 et 127. Cet ensemble est visualisé à une résolution de 1 pixel. Sans artefact de codage, il représente un carré rempli (il n'y a pas d'interstices visibles entre 2 points). Les coordonnées des points du carré, représentables sans perte sur 7 bits, sont quantifiés selon la norme BIFS en utilisant [*vmin*, *vmax*] = [0, 127], pour des valeurs de *N* variant de 8 à 1. La Figure 3.5 montre les résultats obtenus.

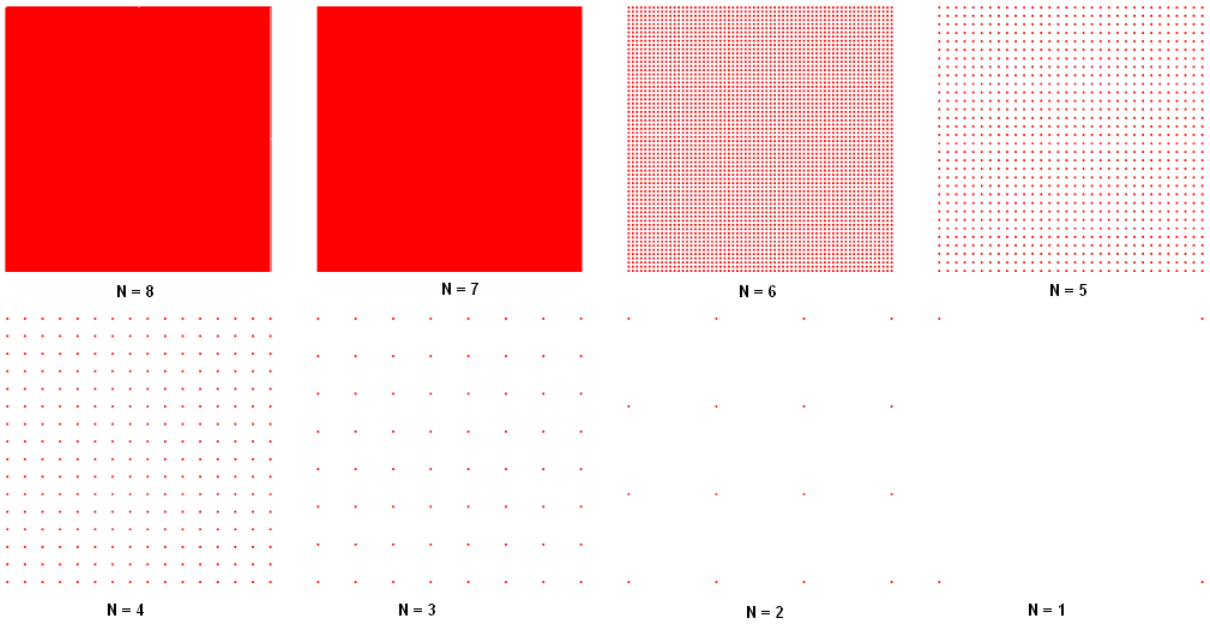


Figure 3.5 – Artefacts de quantification BIFS

On remarque que les carrés, pour lesquels *N* est supérieur ou égal à 7, ne présentent pas d'artéfacts visibles de codage, car les 2^N valeurs entières codables suffisent pour représenter les 128 valeurs possibles du quadrillage. De plus, l'affichage avec anti-crénelage permet de voir un résultat correct même si les points ne sont plus, après décodage, alignés sur des pixels entiers. Par contre, si aucun anti-crénelage n'est utilisé, même pour *N* égal à 8 des artefacts de codage ont lieu, comme le montre la Figure 3.6 : certaines valeurs après décodage ne correspondent pas à des entiers et l'anticrénelage peut faire disparaître certains pixels.

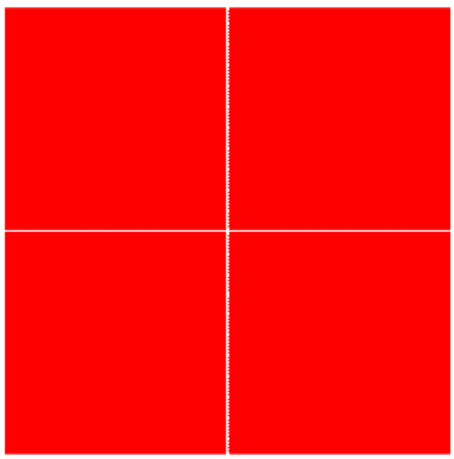


Figure 3.6 – Artefacts de quantification BIFS sans anti-crénelage (N=8)

En revanche, si *N* est inférieur à 7, des artefacts apparaissent, même avec anti-crénelage. La déformation des points est régulière. Les points s'agglutinent sur une grille précise. Cette grille correspond aux 2^N points, à coordonnées décimales, équi-répartis entre *vmin* et *vmax*, inclus. Les

autres points sont agrégés sur cette grille par la fonction d'arrondi ou de troncature selon le choix de l'encodeur.

3.3.2.3 Codage arithmétique BIFS

Enfin, la norme BIFS permet d'utiliser une technique de codage arithmétique pour profiter de la redondance dans les listes de valeurs décimales. Pour coder une telle liste, la norme spécifie un processus en trois étapes. Tout d'abord, une passe de quantification, décrite dans la partie précédente, est appliquée à toutes les valeurs. A la fin de cette étape, il reste à coder une liste d'entiers. Une passe de prédiction, c'est-à-dire de calcul des différences, entre les valeurs quantifiées (entières) successives est effectuée. Enfin, une phase de codage arithmétique est appliquée sur les valeurs prédites. Ce processus est décrit dans la Figure 3.7.

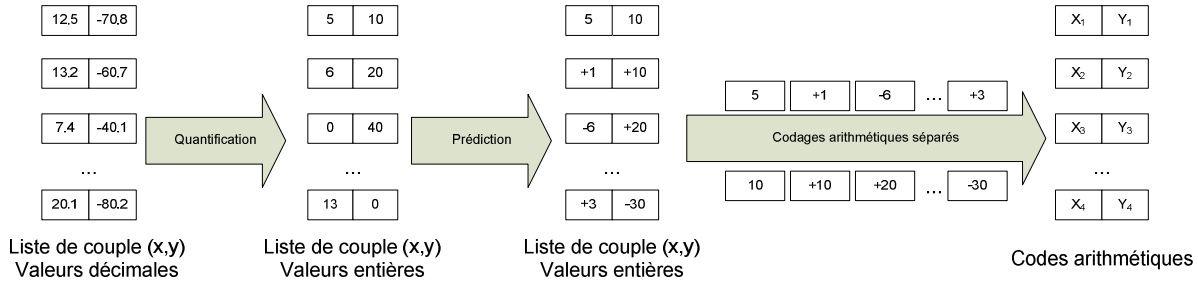


Figure 3.7 – Description du processus de codage arithmétique

Le postulat pour l'efficacité de cette technique est que la dynamique des différences entre les valeurs quantifiées est plus faible que la dynamique des valeurs quantifiées elles-mêmes. Il est également possible d'indiquer des index dans la liste, où la prédiction ne sera pas faite par rapport à la valeur précédente mais par rapport à la valeur zéro. Ceci est utile dans le cas où la différence entre deux valeurs quantifiées successives serait trop grande et aggraverait les performances du codage arithmétique, car elle nécessiterait un trop grand nombre de bits.

3.3.2.4 Quantification LASeR

SVG étant un langage de description XML, il n'offre pas de techniques de compression particulières. La norme indique uniquement qu'il est possible de compresser des fichiers SVG en utilisant le format GZIP. En revanche, la norme MPEG-4 LASeR utilise des techniques de codage spécifiques pour la compression de données SVG. Ces techniques sont dérivées des techniques utilisées en BIFS et en Flash, notamment la quantification.

La quantification LASeR est très proche de celle BIFS. Elle permet de coder des valeurs décimales sur N bits, avec une résolution de $2^{-}r$, fixe pour toute la scène, comme en Flash. La formule de codage est la suivante :

$$v_q = rel_N\left(v \cdot 2^r\right)$$

où *relN*() est une fonction qui arrondi (ou tronque) un nombre décimal en nombre entier relatif et le représente sur *N* bits. La reconstruction est très simple comme l'indique la formule suivante :

$$\hat{v} = v_q \cdot 2^{-r}$$

Un exemple de codage est illustré dans la Figure 3.8.

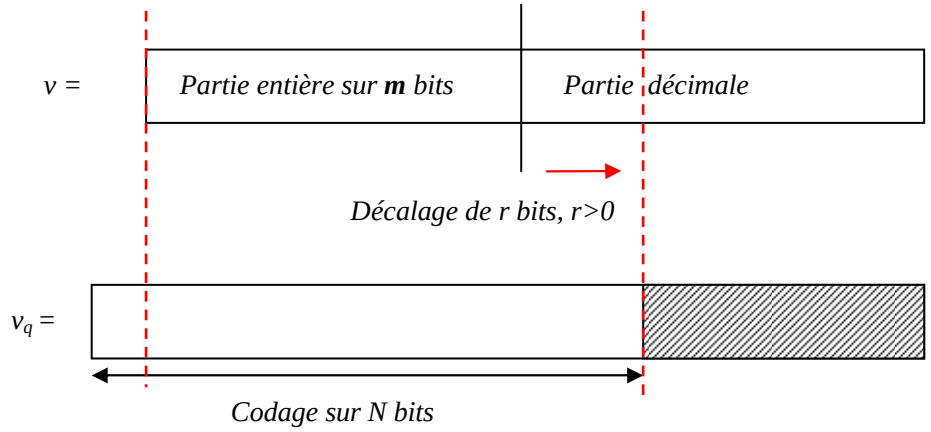


Figure 3.8 – Codage d'une valeur décimale en LASeR

Prenons *v* égal à 7.0, *m* égal à 3, *r* égal à 3 et *N* égal à 6. Supposons une représentation des nombres décimaux en point fixe 8.8 et une représentation des entiers sur 16 bits.

v vaut alors en binaire 00000111.00000000 et v_q vaut 111000.

Pour obtenir ce résultat, il faut décaler *v* à droite de 5 bits (8–r), considéré le résultat comme un entier et ne prendre que les *N* bits de poids faible.

On voit, au travers cette figure et cet exemple, qu'il existe en réalité une relation entre *N*, *m* et *r*, sous la forme d'une contrainte implicite :

$$N \geq m + r$$

En effet, pour avoir un codage significatif, cette équation traduit que les bits de poids fort ne doivent pas être perdus.

3.3.2.5 Synthèse sur les mécanismes de quantification

Les différences avec la quantification à la BIFS sont notables : la non utilisation des bornes *vmin*, *vmax*, et l'utilisation d'une puissance de 2 comme facteur multiplicateur de la valeur à coder. Ces différences ont pour avantage de pouvoir effectuer le codage et le décodage par des opérations de décalages de bits, très simples à réaliser, comme illustré précédemment.

La méthode de quantification LASeR présente plusieurs inconvénients :

- les valeurs à coder doivent être centrées autour de 0 pour reproduire l'équivalent de *vmin*;

- et, l'absence d'utilisation de la valeur *vmax* implique une moins bonne précision de quantification.

3.3.2.5.1.1 Centrage des valeurs

Le premier inconvénient peut, dans certains cas, être évité en soustrayant la valeur (*vmin*- *vmax*)/2 à toutes les valeurs à coder, et en modifiant la scène, en insérant une translation de (*vmin*+ *vmax*)/2. Les valeurs de translation étant codées en utilisant le même nombre de bits que les coordonnées, cette méthode n'est efficace que si, le codage de la translation supplémentaire ne nécessite pas d'augmenter *N*, c'est-à-dire si :

$$\log_2\left(\frac{v_{\min}+v_{\max}}{2}\right) \le \log_2\left(\max\left(|v_{\min}|,|v_{\max}|\right)\right)$$

De plus, l'ajout de cette translation modifie la structure de la scène, donc peut être incompatible avec les scripts. Enfin, la soustraction de (v_{min}- v_{max})/2 à toutes les valeurs est difficilement faisable pour les valeurs d'animations ou les valeurs contenues dans les scripts, car une même valeur peut avoir différentes utilisations au sein d'un script.

3.3.2.5.1.2 Différence de précision

Le second inconvénient ne peut malheureusement pas être évité. Il peut être illustré en prenant un exemple. Considérons des coordonnées à coder entre $v_{\min}$=0 et $v_{\max} > v_{\min}$. La formule de quantification selon la norme BIFS devient donc très proche de celle issue de la norme LASeR :

$$v_q = \mathrm{int}\left(v\frac{2^N-1}{v_{\max}}\right)$$

La valeur décodée en BIFS sera :

$$\hat{v} = \mathrm{int}\left(v\frac{2^N-1}{v_{\max}}\right)\frac{v_{\max}}{2^N-1}$$

Selon le codage BIFS, il y aura donc 2^N valeurs codées, occupant tout l'espace entre 0 et $v_{\max}$. Selon le codage LASeR, en omettant le cas des nombres négatifs, la formule de reconstruction d'un nombre entier codé sur N bits est :

$$\hat{v} = \mathrm{int}_N\left(v\cdot 2^r\right)\cdot 2^{-r}$$

Donc, le codage LASeR produira 2^N valeurs comprises entre 0 et 2^{N-r}. Or la valeur 2^{N-r} étant supérieure ou égale à $v_{\max}$, mis à part le cas d'égalité, le codage LASeR permet inutilement de coder des valeurs qui sont supérieures à la valeur maximale utilisée. Il en résulte que, pour un même nombre de bits *N* utilisé pour coder une plage de valeurs, la distance entre 2 points de la grille de quantification BIFS est

plus faible que la distance entre 2 points de la grille de quantification LASeR. En d'autres termes, la résolution en BIFS sera meilleure que la résolution LASeR.

Examinons le cas où $v_{min} = 0$ et $v_{max} = 768$. Il faut au moins 10 bits dans les 2 cas (BIFS et LASeR) pour représenter la partie entière de v_{max}. Pour $N = 10$, la distance entre 2 points de la grille BIFS sera 768/1024 soit 0,75 pixels alors qu'en LASeR, la distance sera de 1 pixel. La Figure 3.9 indique, en ordonnée, la distance minimale, en pixels, entre les valeurs issues du décodage des nombres compris entre 0 et 768, en fonction du nombre de bits *N*, en abscisse, utilisé par la quantification BIFS (courbe pointillée) et LASeR (courbe pleine). Cette figure illustre le fait que la quantification BIFS est systématiquement plus fine que la quantification LASeR.

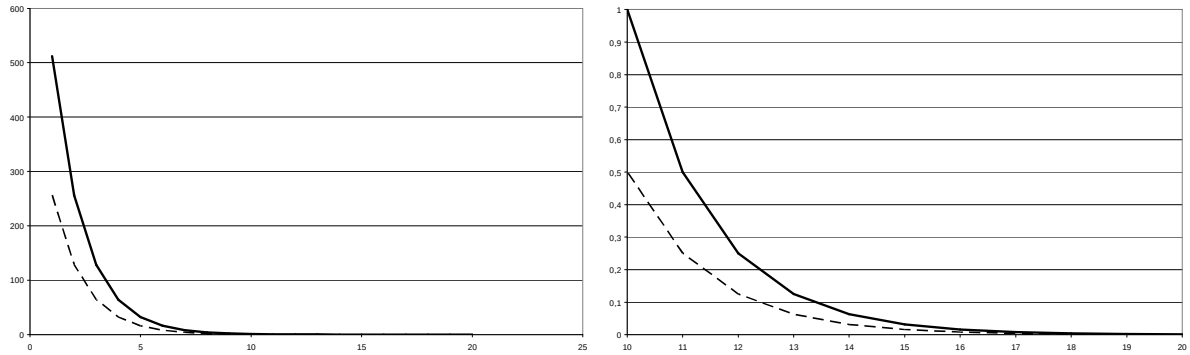


Figure 3.9 – Comparaison de la précision (en pixels) de quantification BIFS et LASeR en fonction du nombre de bits utilisés

3.3.2.5.1.3 Granularité de la signalisation

Enfin, le dernier inconvénient du codage LASeR par rapport au codage BIFS provient de la granularité de la signalisation de la quantification. Alors qu'en BIFS, il est possible de quantifier nœud par nœud, avec un surcoût dû à la signalisation, en LASeR, toute la scène doit être quantifiée avec les mêmes paramètres. Cela impose de créer tous les dessins dans un même repère de construction avec la même précision (de manière similaire à la construction d'une fonte où tous les glyphes sont conçus dans un même repère, la « em box »), et d'ajouter des facteurs de zoom aux différents objets pour leur donner les proportions relatives les uns par rapport aux autres.

3.3.3 Compression des objets graphiques

Les gains en compression sur les descriptions de scènes sont fortement liés aux choix de représentation des objets graphiques de la scène. Dans le chapitre 1, nous avons présenté deux méthodes de représentation d'objets graphiques : la représentation en contour et celle en carte planaire. Nous voyons ici l'impact de ces représentations notamment sur la compression. Dans les deux types de représentations, les informations à coder sont généralement : des points, des couleurs, des épaisseurs et des commandes de tracé.

3.3.3.1 Représentation binaire des cartes planaires Flash

La particularité de la représentation sous forme de carte planaire qu'utilise le format SWF est que les points n'appartenant qu'à un seul tracé ne sont pas répétés. On parcourt un tracé une seule fois en indiquant la couleur à droite et la couleur à gauche. En revanche, on doit signaler potentiellement un grand nombre de fois, les couleurs de remplissage. En effet, il faut signaler les nouvelles couleurs de tracé à chaque fois que l'on croise un autre tracé dont les couleurs sont différentes. Par exemple, sur la Figure 1.6, il faut signaler des changements de couleurs au point 1 et au point 2. Cette figure nécessite ainsi d'indiquer deux fois chaque couleur, sachant que pour des tracés consécutifs ayant la même propriété de couleur, ces indications de couleurs ne sont pas répétées.

Les points dans le format SWF sont codés sous forme différentielle et sur un nombre variable de bits, alors que les couleurs sont codées en utilisant des index sur une palette de couleurs déclarée au début de l'objet. L'utilisation d'index aboutit ainsi à un codage plus efficace du fait des répétitions des couleurs.

3.3.3.2 Compression de contours

Pour coder une représentation sous forme de contours, il faut à nouveau coder des points, des couleurs et des tracés. Le codage des couleurs est plus direct que précédemment, car chaque contour définit tout au plus deux couleurs (remplissage et tracé). Lorsqu'on utilise cette représentation, si une même couleur n'est utilisée que par un seul contour, il ne faut la coder qu'une seule fois. L'utilisation d'une palette de couleurs est utile uniquement dans le cas où plusieurs contours utilisent la même couleur. Une des limitations de ce modèle est qu'elle force à créer une primitive (fermée ou ouverte) pour chaque trait dont le couple (épaisseur, couleur) est différent.

Pour le codage des points, des différences existent par rapport au codage de représentation planaire. Les points sont codés dans un contour particulier, éventuellement en utilisant un codage différentiel. Mais, sauf pour les points des contours extérieurs à l'objet composite et les points des tracés non fermés, un point participe souvent à au moins deux primitives de tracés. Dans la Figure 3.10, les points du tracé commun (tracé épais) participent ainsi à 2 contours.

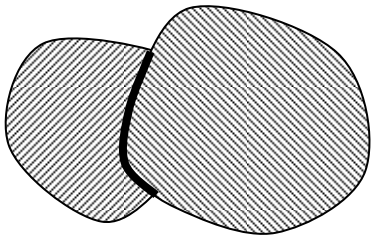


Figure 3.10 – Redondance des points dans une représentation par contours

Cette représentation induit donc une répétition des points par rapport à la représentation en carte planaire. Le Tableau 3.1 indique pour certaines séquences de dessins animés le nombre de points codés, signalés, en utilisant une représentation par carte planaire et par contours. En moyenne, la représentation utilisant des contours ouverts et fermés doit coder 77% de points en plus que la représentation en carte planaire.

Séquence	Représentation par carte planaire	Représentation par contours fermés et ouverts	Facteur de répétition des points
1	69 600	94 357	1,36
1a	76 018	123 036	1,62
2	37 057	56 267	1,52
2a	71 224	129 673	1,82
3	36 402	66 718	1,83
3a	69 622	108 705	1,56
4	43 855	59 191	1,35
4a	45 388	69 460	1,53
5	64 247	90 556	1,41
5a	55 246	79 956	1,45
6	18 980	23 581	1,24
6a	51 576	73 844	1,43
7	27 672	38 367	1,39
7a	33 369	46 228	1,39
8	23 853	29 669	1,24
8a	12 944	37 528	2,90
9	25 299	49 499	1,96
9a	159 501	300 201	1,88
10	2 580	4 326	1,68
11	16 891	41 137	2,44
12	15 585	55 547	3,56
13	47 021	84 902	1,81
14	25 568	64 910	2,54
15	13 175	20 967	1,59
Total	1 042 673	1 748 625	1,68
Moyenne	43 445	72 859	1,77

Tableau 3.1 – Nombre de points codés en fonction de la représentation graphique

Sachant que le codage d'un index est en général moins coûteux que le codage d'un point, même quantifié, il convient donc dans le cas d'une représentation par contour de coder des index de points à la place de coder un même point de multiples fois. Nous verrons dans le chapitre 6, l'impact de cette remarque sur les contenus BIFS.

3.4 Conclusion

Dans ce chapitre, nous avons présenté les différentes techniques existantes pour compresser les descriptions de scènes. Certaines techniques s'attachent à la représentation binaire compacte de la

structure, et dans le cas particulier des structures décrites en XML, des techniques génériques existent. Pour la compression des données, nous avons également passé en revue les techniques existantes, notamment les méthodes de quantification qui s'avèrent très efficaces mais pour lesquelles nous avons décrit les problèmes de dégradation qu'elles peuvent entraîner. Nous aborderons, dans le chapitre 6, le coût lié au traitement de ces algorithmes de codage/décodage.

Chapitre 4 Descriptions de scènes et interactivité

4.1 Introduction

Afin de présenter, dans ce document, une analyse de l'ensemble des mécanismes offerts par les descriptions de scène, nous nous devons de donner, dans ce chapitre, une analyse du fonctionnement des mécanismes d'interactivité, au-delà de la navigation, utilisés par les différents langages de descriptions de scènes. Nous espérons que cette analyse permettra d'appréhender les difficultés sous-jacentes à l'édition de scènes interactives. Enfin, nous présentons également les principaux scénarios actuels exploitant ces mécanismes d'interactivité.

4.2 Les mécanismes d'interactivité

L'interactivité au sens informatique est la faculté d'échange entre l'utilisateur d'un système informatique et la machine, par l'intermédiaire, par exemple, d'un terminal doté d'un écran de visualisation. Cette définition s'applique également aux systèmes informatiques multimédia. On parle ainsi de média interactif ou de « Rich Media » pour parler d'une présentation multimédia interactive. Nous avons vu dans le chapitre 1 que la description d'une scène indiquait les caractéristiques d'une présentation multimédia. Il est donc nécessaire qu'elle décrive également son caractère interactif, c'est-à-dire la capacité d'une scène à réagir aux actions de l'utilisateur et à se modifier en réponse à ces actions.

Nous appellerons "modèle évènementiel", l'ensemble des processus mis en œuvre pour le traitement d'une action utilisateur. Chaque langage de descriptions de scènes se doit de décrire son modèle évènementiel. Cependant, on peut tenter de décrire un modèle événementiel unifié, comme suit. Ce modèle se découpe en quatre phases :

a. capture de l'action utilisateur par l'application et génération d'un évènement selon le format de la scène,

b. transmission de l'évènement à l'élément concerné dans la scène,

c. utilisation de l'évènement et modification de la scène,

d. et transmission éventuelle d'évènements résultants.

L'enchaînement de ces phases est illustré dans la Figure 4.1. Nous détaillons ces différentes phases dans la suite de ce chapitre.

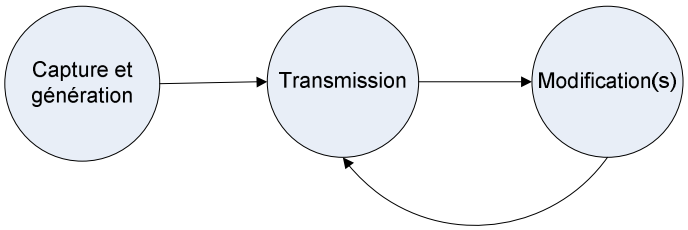


Figure 4.1 – Diagramme simplifié du traitement d'une action utilisateur

4.2.1 Détection des évènements utilisateurs

4.2.1.1 Capture des évènements utilisateurs

Dans les systèmes informatiques, les évènements utilisateurs sont captés par le système d'exploitation et sont ensuite transmis à l'application. Dans le cas qui nous concerne l'application est le lecteur de contenu multimédia interactif. Il existe de nombreux évènements utilisateurs captables par le lecteur et utilisables dans une scène multimédia. Le type de ces événements est souvent dépendant du terminal sur lequel le contenu est présenté. Parmi les évènements possibles, on peut remarquer les périphériques et évènements suivants :

- Si le terminal possède un clavier, on peut généralement capter des évènements de type : touche enfoncée, touche relâchée, combinaison de touches et, après application de la configuration du clavier (langue, mode de saisie), obtenir le caractère correspondant à la touche. Cependant, tous les terminaux ne possèdent pas les mêmes claviers. Les différences entre les claviers sont principalement le nombre et la nature des touches. Il est donc difficile de concevoir une scène unique pour tous les terminaux. On peut, par exemple, citer les touches "SoftKey1" et "SoftKey2" présentes et prédominantes pour l'interactivité sur tous les téléphones mobiles mais qui n'existent pas sur les claviers de PC actuels.
- Si le terminal possède une souris, on peut obtenir des évènements du type : bouton appuyé, bouton relâché, bouton appuyé et relâché rapidement (simple clic), ou appuyé et relâché deux fois (double clic), déplacement de la souris, combinaison d'un ou plusieurs boutons appuyés ou relâchés et du déplacement (appelé évènement glisser-déposer, en anglais *drag-and-drop*).
- Si le terminal possède un stylet, les évènements captables sont : stylet appuyé, relâché, clic et double clic, déplacement quand le stylet est appuyé. Il n'est pas possible, en général, de détecter le mouvement quand le stylet n'est pas appuyé.

Enfin, il existe également des terminaux connectés à des périphériques ad hoc (joystick, télécommandes, périphérique haptique, olfactif …) pour lesquels il faut concevoir des scènes spécifiques.

Le problème principal posé par cette phase de capture est donc la dépendance au terminal. En effet, créer du contenu interactif qui soit jouable sur des terminaux divers est une tâche compliquée. Il faut prendre en compte que certaines interactions ne sont possibles que sur certains types de terminaux.

4.2.1.2 Traduction des évènements utilisateurs en évènements de scène

La phase de capture traduit les actions (ou événements) de l'utilisateur en évènements internes à la scène. Les langages de descriptions de scènes définissent des structures qui indiquent quand un évènement doit être capté et comment il doit être traduit dans le format de la scène. Pour ces structures, on parle d'écouteur d'évènements (*listener*). L'écouteur est l'entité responsable d'écouter un évènement particulier. Les actions utilisateurs n'ont aucun impact sur la scène si aucun écouteur n'est présent. Un écouteur peut être attaché à la scène complète ou à un élément particulier de la scène, quand l'événement ne doit être capté que dans le contexte particulier lié à cet élément.

Les écouteurs d'évènements peuvent être attachés à un objet (ou à la scène entière), de manière déclarative ou programmatique. Un exemple simple est le traitement d'un clic de la souris. On peut écouter les clics souris globalement au niveau de la scène, mais on les écoute généralement sur un objet particulier (un bouton). De même, le traitement des évènements clavier peut se faire au niveau d'un objet (l'objet graphique qui possède le focus, c'est-à-dire qui est sélectionné) ou au niveau de la scène. Les exemples Code 4.1, Code 4.2, Code 4.3 et Code 4.4 illustrent ces différentes possibilités dans différents langages.

```
Transform {
  children [
    Shape { … }
    Shape { … }
    TouchSensor {}
  ]
}
```

Code 4.1 – Déclaration d'un écouteur de la souris dans le langage VRML

Dans le cas de VRML (Code 4.1), l'association d'un écouteur de la souris sur des objets visuels s'effectue en plaçant, dans l'arbre de scène, un nœud `TouchSensor` au même niveau que les nœuds `Shape` représentant les objets visuels.

```
<xmlev:listener event="click" observer="a" target="b" … />
```

Code 4.2 – Déclaration d'un écouteur du clic de la souris dans le langage XML Events

Dans le langage *XML Events* [67] (Code 4.2), la déclaration d'un écouteur d'évènement peut être faite à n'importe quel endroit dans l'arbre DOM, à l'aide d'un élément `listener`, déclaré dans l'espace de nom réservé par la spécification XML Events (ici `'xmlev'`). L'écouteur indique l'endroit où il écoute, ici l'objet `a`, grâce à l'attribut `observer`, et le type d'évènement écouté, ici le clic, grâce à l'attribut `event`. L'attribut `target` peut être important à remarquer : il identifie un nœud dans l'arbre de scène où l'évènement est déclenché. En effet, le modèle DOM permet de capter un évènement à partir d'un endroit différent de la source. Nous détaillerons ce procédé dans la section 4.2.2.

```
<svg onload="init();" … >
<rect id="a" …/>
<script type="application/ecmascript">
function init() {
  document.getElementById('a').addEventListener('click', act, true);
}
function act() {…}
</script>
</svg>
```

Code 4.3 – Ajout d'un écouteur DOM par programmation ECMAScript

Les interfaces DOM (Code 4.3) permettent d'ajouter des écouteurs d'évènements en utilisant la fonction `addEventListener` par exemple dans le langage ECMAScript. L'utilisation de langage de script est équivalente à l'utilisation de la représentation XML dans l'exemple précédent.

```
function add():MovieClip {
      var mc:MovieClip = _root.createEmptyMovieClip("myMC", 0);
      mc.onRollOver = function() { act(); };
      return mc;
}
function act() {…}
```

Code 4.4 – Ajout d'un écouteur Flash par programmation ActionScript 3

Dans le langage Flash (Code 4.4), certains objets ont la capacité d'écouter des évènements. Les objets `Button` et `MovieClip` réagissent par exemple aux mouvements de la souris. Le langage ActionScript permet ensuite de définir le traitement qui sera effectué au moment de l'action utilisateur. Dans cet exemple, la fonction `act` sera appelée quand l'utilisateur survolera l'objet avec sa souris.

Certaines phases de capture peuvent être plus coûteuses en ressource que d'autres. Par exemple, à chaque évènement souris reçu, il faut parcourir l'ensemble des objets sensibles au clic (c'est-à-dire auquel un écouteur est attaché) et déterminer si le clic a eu lieu au dessus de l'un de ces objets. Cette opération, appelée le *picking*, implique notamment des calculs matriciels pour transformer les coordonnées du pointeur de la souris du système de coordonnées de la fenêtre de visualisation au système de coordonnées de l'objet. Elle peut impliquer, si l'objet à une forme non rectangulaire, des calculs d'appartenance d'un point à une forme géométrique et donc s'avérer relativement coûteuse en

calcul. A l'inverse, la capture de la pression d'une touche du clavier est un processus relativement trivial.

A partir de cette présentation du fonctionnement de la phase de traduction d'évènements utilisateurs en évènement scène, on peut retenir qu'il faut donc concevoir les scènes interactives en prenant en compte, non seulement les interactions possibles sur les terminaux cibles, mais également la puissance nécessaire pour traiter chaque évènement, en particulier pour les terminaux à faible capacité de calcul (comme les terminaux mobiles).

4.2.2 Transmission des évènements

Une fois l'évènement capté par un écouteur, celui-ci est ensuite transmis à une ou plusieurs entités chargées du traitement de cet évènement. On appelle ces entités des gestionnaires d'évènements ou *handler*. Il existe différentes méthodes pour signaler l'association écouteur-gestionnaire.

Dans le langage VRML, cette association est faite de manière explicite à l'aide d'une route, selon le mécanisme décrit dans le chapitre 2, reliant écouteur et gestionnaire. Si plusieurs gestionnaires doivent traiter le même évènement, la scène comportera plusieurs routes.

Dans les langages XML, l'élément `listener`, vu précédemment, possède un attribut `handler` qui indique l'élément qui traitera cet évènement. Il peut s'agir d'une animation (pour l'activer), d'un hyperlien (pour déclencher la navigation) ou d'un script (pour l'exécuter). Pour qu'un même évènement soit traité par plusieurs gestionnaires, il faut également utiliser plusieurs éléments `listener`.

La transmission d'un évènement jusqu'au gestionnaire est simple en général. Cependant, le processus se complique si le traitement d'un évènement dépend de l'endroit dans l'arbre de scène où il est traité, ou, si plusieurs éléments sont susceptibles d'être intéressés par un même évènement. En effet, si un seul nœud est intéressé par un évènement précis, il n'est pas nécessaire de propager l'évènement après traitement par le gestionnaire. En revanche, si plusieurs capteurs d'un même évènement sont présents dans la scène, il faut connaître l'ordre dans lequel seront effectués les traitements de cet évènement au niveau des différents écouteurs. Supposons qu'une scène contienne un sous-arbre qui, si l'on clique dessus, déclenche le changement de la scène. Supposons encore que dans ce sous-arbre, un objet soit déplaçable avec la souris. Un clic sur cet objet pourra produire des résultats différents si l'évènement est traité d'abord par la logique de changement de scène ou par la logique de déplacement de l'objet. Les langages de descriptions de scènes spécifient donc l'ordre de propagation des évènements.

Le standard W3C *DOM Events* définit trois méthodes/phases de propagation des évènements dans un arbre de scène :

- la capture (en anglais *capture*), qui consiste à parcourir l'arbre en descendant du nœud racine vers les nœuds feuilles et à transmettre à chaque nœud intéressé l'évènement en question ;

- le ciblage (en anglais *target*), qui consiste à transmettre l'évènement uniquement au nœud concerné sans le transmettre ni à ses parents, ni à ses enfants ;
- et le bouillonnement (en anglais *bubble*), qui consiste à démarrer du nœud issu de la phase de ciblage et à remonter l'arbre en transmettant l'évènement aux nœuds parents.

Le modèle évènementiel DOM Events permet à un même évènement d'être traité dans ces trois phases dans l'ordre suivant : capture, ciblage, bouillonnement. Certains évènements peuvent être restreints par définition à pas subir la phase de bouillonnement. Enfin, dans ce modèle, chaque écouteur s'enregistre pour une phase précise et à la possibilité d'arrêter la propagation d'un évènement.

Le standard VRML (BIFS), en revanche, définit une seule méthode de propagation des évènements en utilisant les routes. La propagation d'évènement selon le modèle DOM n'existe pas. Cependant, il est possible d'émuler cette propagation en multipliant le nombre de routes.

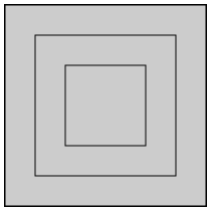


Figure 4.2 – Bouillonnement d'évènement (a)

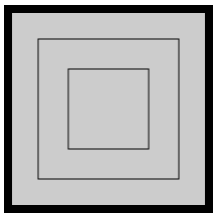


Figure 4.3 – Bouillonnement d'évènement (b)

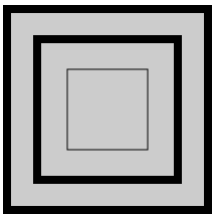


Figure 4.4 – Bouillonnement d'évènement (c)

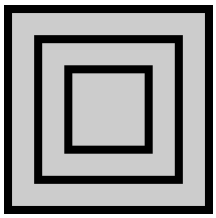


Figure 4.5 – Bouillonnement d'évènement (d)

Les figures ci-dessus (Figure 4.2, Figure 4.3, Figure 4.4 et Figure 4.5) montrent quatre états d'une même scène, décrite en SVG dans le Code 4.5, qui tire profit de la propagation ascendante par défaut. Cette scène comprend trois rectangles visuellement imbriqués. L'état (a) décrit la scène lorsque la souris ne survole aucun objet. L'état (b) décrit la scène quand la souris survole le rectangle le plus extérieur. Seule l'apparence de ce dernier est modifiée (contour plus épais) car le bouillonnement n'a pas d'impact du fait que le rectangle est dans l'élément g le plus englobant. L'état (c) décrit la scène quand le rectangle du milieu est survolé. La propagation ascendante, par défaut, de l'évènement "entrée dans l'objet" conduit à la modification des deux rectangles extérieurs. Enfin, l'état (d) décrit la scène quand le rectangle le plus imbriqué est survolé. La propagation ascendante conduit dans ce cas à la modification des trois rectangles.

```
<svg onload="init_menu()" fill="#ccc" stroke="black" stroke-width="1">
 <g id="element1">
  <rect x="5" y="5" width="200" height="200" />
  <g id="element2">
   <rect x="35" y="35" width="140" height="140" />
   <g id="element3">
    <rect x="65" y="65" width="80" height="80"/>
   </g>
  </g>
 </g>
 <script type="text/ecmascript">
 function init_menu() {
  document.getElementById("element1").addEventListener('mouseover', entrée,
   false);
  document.getElementById("element2").addEventListener('mouseover', entrée,
   false);
  document.getElementById("element3").addEventListener('mouseover', entrée,
   false);
  document.getElementById("element1").addEventListener('mouseout', sortie,
   false);
  document.getElementById("element2").addEventListener('mouseout', sortie,
   false);
  document.getElementById("element3").addEventListener('mouseout', sortie,
   false);
 }

 function entrée(evt) {
  evt.currentTarget.setAttributeNS(null, "stroke-width", "8");
 }

 function sortie(evt) {
  document.getElementById("element1").setAttributeNS(null, "stroke-width",
   "1");
  document.getElementById("element2").setAttributeNS(null, "stroke-width",
   "1");
  document.getElementById("element3").setAttributeNS(null, "stroke-width",
   "1");
 }
]]>
</script>
</svg>
```

Code 4.5 – SVG et le bouillonnement d'évènements

On remarque plusieurs points importants dans le Code 4.5. Tout d'abord, au moment du chargement du fichier (`onload`), deux écouteurs sont associés, par la fonction `addEventListener`, à chaque élément `g` : un écouteur pour l'évènement "entrée de la souris" associé à la fonction du script intitulée `entrée`, et un autre écouteur pour l'évènement "sortie de la souris" associé à la fonction `sortie`. La fonction `entrée` utilise l'endroit courant dans l'arbre de scène où l'évènement se trouve (`currentTarget`) pour effectuer la modification de la scène. Il est intéressant de remarquer qu'aucun élément de syntaxe n'indique le bouillonnement des évènements, qui pourtant aura bien lieu.

Le Code 4.6 montre comment exprimer cette scène en BIFS suivant le modèle évènementiel VRML.

```
Layer2D {
 children [
  DEF G1 Transform2D {
   translation -240 150
   children [
    DEF TS1 TouchSensor {}
    DEF S1 Shape {
     geometry Rectangle { size 100 150 }
     appearance Appearance {
      material Material2D {
       filled TRUE
       lineProps DEF L1 LineProperties { width 1 }
      }
     }
    }
    DEF G2 Transform2D {
     translation 100 -50
     children [
      DEF TS2 TouchSensor {}
      DEF S2 Shape {
       geometry Rectangle { size 100 150 }
       appearance Appearance {
        material Material2D {
         filled TRUE
         lineProps DEF L2 LineProperties { width 1 }
        }
       }
      }
      DEF G3 Transform2D {
       translation 100 -50
       children [
        DEF TS3 TouchSensor {}
        DEF S3 Shape {
         geometry Rectangle { size 100 150 }
         appearance Appearance {
          material Material2D {
           filled TRUE
           lineProps DEF L3 LineProperties { width 1 }
          }
         }
        }
       ]
      }
     ]
    }
   ]
  }
  DEF C1 Conditional {buffer{REPLACE L1.width BY 10 }}
  DEF C2 Conditional {buffer{REPLACE L2.width BY 10 }}
  DEF C3 Conditional {buffer{REPLACE L3.width BY 10 }}
  DEF RC Conditional{
   buffer{
     REPLACE L1.width BY 1
     REPLACE L2.width BY 1
     REPLACE L3.width BY 1
   }
  }
]
}
#Routes pour la propagation directe
ROUTE TS1.isOver TO C1.activate
```

```
ROUTE TS2.isOver TO C2.activate
ROUTE TS3.isOver TO C3.activate
ROUTE TS1.isOver TO RC.reverseActivate
ROUTE TS2.isOver TO RC.reverseActivate
ROUTE TS3.isOver TO RC.reverseActivate

#Routes pour la propagation ascendante
ROUTE TS3.isOver TO C2.activate
ROUTE TS3.isOver TO RC.reverseActivate
ROUTE TS3.isOver TO C1.activate
ROUTE TS3.isOver TO RC.reverseActivate
ROUTE TS2.isOver TO C1.activate
ROUTE TS2.isOver TO RC.reverseActivate
```

Code 4.6 – BIFS et le bouillonnement d'évènements

Outre le fait que l'on n'ait pas utilisé de script dans cette version mais des nœuds BIFS du type `Conditional`, on remarque qu'il existe autant de routes pour la propagation directe de l'évènement que d'écouteurs créés dans la scène SVG. Cependant, pour la propagation ascendante, des routes supplémentaires sont nécessaires. Dans ce cas, il en faut $N \times (N-1)$, où N représente la profondeur de l'arbre de scène, ici 3. A partir de cet exemple, on se rend compte, qu'il est possible de représenter le mécanisme de propagation ascendante du modèle DOM Events à l'aide de routes dans le modèle VRML, mais avec la difficulté liée au nombre de routes. Une remarque similaire peut être faite pour la propagation descendante. On peut aussi noter la difficulté de représenter de manière souple la possibilité du modèle DOM Events d'annuler la propagation d'un évènement.

4.2.3 Cascade d'évènements et gestion temporelle

Dans les descriptions de scènes complexes, un évènement déclenche de nombreuses modifications de la scène. En fonction de l'action utilisateur, il faut effectuer des traitements, par exemple en utilisant une fonction script ou un nœud du type `Conditional`, pour appliquer les modifications sur la scène. Le résultat de ces traitements peut déclencher d'autres évènements. Pour caractériser la multiplication des évènements successifs, consécutifs à une seule action de l'utilisateur, on parle d'une cascade d'évènements primaires, secondaires … en fonction du niveau de l'évènement dans la cascade.

La façon de traiter cette cascade dans une implémentation peut avoir un impact important sur la scène en termes d'aspect visuel ou de performance. Si cette cascade d'évènements primaires, secondaires, tertiaires … est traitée de manière atomique, c'est-à-dire si tous les évènements de la cascade sont traités avant de produire un nouvel affichage de la scène, les conséquences de l'action utilisateur pourront nécessiter un temps important pour être visualisées par l'utilisateur. A l'inverse, si un nouvel affichage est produit avant le traitement complet de la cascade, le résultat de l'action utilisateur sera perçu progressivement comme une série de modifications de la scène appliquées à différents instants dans le temps. Cette seconde approche peut être problématique si l'utilisateur interagit à nouveau avec la scène alors que toutes les conséquences de l'interaction précédente ne sont pas appliquées. Le traitement atomique est l'option généralement choisie par la plupart des implémentations actuelles.

De même, la gestion de la cascade d'évènements impose des contraintes sur l'exécution des scripts ou de nœud `Conditional`. En effet, un script (ou un nœud `Conditional`) comporte souvent plusieurs modifications de la scène (ajout d'un élément, modification d'attributs …). Il existe deux méthodes pour les appliquer et déclencher les évènements suivants dans la cascade : soit les évènements liés à une modification de la scène sont générés immédiatement après la modification, soit ils sont générés à la fin de l'exécution du script ou du nœud `Conditional`. Le choix d'une méthode ou d'une autre pourra ne pas produire le même résultat. En général, les langages rencontrés utilisent globalement la première méthode, car la seconde méthode nécessite d'empiler les évènements pendant l'exécution du script.

Un second problème du traitement de cette cascade est lié à l'ordre dans lequel sont traités les évènements secondaires. Supposons qu'un évènement primaire entraîne la création de deux évènements secondaires. Dans quel ordre traiter ces nouveaux évènements ? Doit-on traiter tous les évènements secondaires avant les évènements tertiaires ou doit-on traiter un évènement tertiaire, conséquence d'un évènement secondaire, avant les autres évènements secondaires ? Autrement dit, si on construit un arbre des évènements, doit-on faire un parcours en profondeur ou un parcours en largeur ? A nouveau, le choix de ce parcours pourra produire des résultats différents. Le langage VRML a fait le choix d'un parcours en largeur. Le modèle DOM Events utilise un parcours en profondeur.

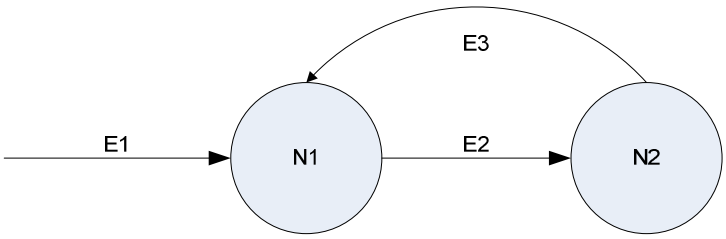


Figure 4.6 – Boucle d'évènement

Un troisième problème qu'il faut traiter dans le modèle évènementiel, lié à la présence d'évènements secondaires, est la gestion des boucles, comme l'illustre la Figure 4.6. En effet, un évènement E_1 pouvant déclencher un autre évènement E_2, il est possible qu'E_2 déclenche à son tour un évènement E_3 du même type que l'évènement E_1, avec les mêmes conséquences. Cela crée une boucle. La méthode utilisée dans le modèle VRML ou le modèle DOM pour casser ces boucles est d'associer à chaque évènement un temps correspondant au cycle de composition dans lequel l'évènement a été généré. Chaque nœud conserve la date du dernier évènement qu'il a traité et s'il reçoit un nouvel évènement avec la même date, une boucle est détectée et le traitement de ce nouvel évènement est ignoré.

Enfin, un quatrième problème, spécifique aux normes VRML et MPEG-4 BIFS, est le problème dit de « fan-in, fan-out ». Comme un nœud peut générer plusieurs évènements dans un même cycle, l'ordre

dans lequel ces évènements sont générés en sortie de nœud est important. C'est le problème de « fan-out ». De même, comme un même nœud peut recevoir plusieurs évènements dans un même cycle, l'ordre de traitement des évènements en entrée est appelé problème de « fan-in ». La spécification VRML bien qu'elle souligne ces problèmes, ne propose pas de solution.

4.2.4 Modification de la scène

Il existe différentes méthodes pour exprimer les modifications qui doivent être appliquées à une scène à la suite d'un évènement (utilisateur ou non). On peut les classer selon deux catégories :

- les méthodes déclaratives, en utilisant un langage textuel ou binaire pour décrire les modifications à appliquer;
- et les méthodes programmatiques, qui utilisent un langage de script.

L'avantage des méthodes déclaratives réside dans le fait que les comportements peuvent être implémentés nativement et donc plus efficacement. Ainsi, si le traitement est simple, le langage déclaratif sera moins coûteux à effectuer. En revanche, si le traitement nécessite des opérations complexes à décrire dans un langage déclaratif (calcul mathématique par exemple) alors le langage de script sera plus approprié. Parmi les méthodes déclaratives pour modifier une scène, on peut citer le mécanisme de « BIFS Updates » [43] et ou celui décrit dans la spécification « REX Events » [59], présentés au chapitre 2.

Cette méthode de modification peut également bien s'intégrer avec les mécanismes d'interactivité précédemment décrits. En effet, la norme MPEG-4 BIFS permet, grâce à l'utilisation d'un nœud particulier, le nœud `Conditional`, d'indiquer une série de mises à jour à exécuter quand ce nœud reçoit un évènement booléen vrai sur son entrée `activate`, ou un évènement faux sur son entrée `reverseActivate`.

La plupart des langages de descriptions de scènes comme VRML, BIFS ou SVG offrent les deux possibilités mais la diversité des modifications possibles, sans recourir à un langage de script, est différente, notamment à cause de la manière de décrire les évènements.

Les modifications qui sont appliquées à la scène à la suite d'un évènement peuvent être de deux types :

- Soit elles dépendent uniquement de la présence de l'évènement et la nature exacte de l'évènement n'est pas importante. C'est par exemple le cas si on souhaite déclencher une action sur pression d'une touche, indépendamment de la touche pressée.
- Soit les modifications de la scène utilisent un ou plusieurs paramètres issus de l'évènement, comme, par exemple, le code de la clé appuyé, la position de la souris au moment du clic…

La plupart des langages de descriptions de scènes permettent déclarativement le premier type de modifications. Pour le second type, il est intéressant de présenter la différence de méthode pour extraire les paramètres issus d'un évènement, entre les modèles évènementiels VRML et DOM.

Dans le langage VRML, les capteurs d'action utilisateur décomposent une action en un ou plusieurs évènements typés simples et exploitables directement par les autres nœuds. Par exemple, grâce au nœud `PlaneSensor2D`, il est possible de récupérer le déplacement (x,y) du pointeur de la souris entre la pression du bouton et son relâchement, sans avoir à utiliser un langage de script. Cela permet notamment de déplacer des objets (*drag and drop*). Le Code 4.7 décrit les différents évènements de sortie du nœud BIFS `PlaneSensor2D`. L'évènement `isActive` indique si une opération de déplacement est en cours. Il est utilisé pour déclencher des modifications du premier type. Les évènements `trackPoint_changed` et `translation_changed` indiquent respectivement la position courante de la souris et le déplacement de la souris depuis que le bouton de la souris a été enfoncé. Les valeurs de ces évènements sont directement utilisables, par exemple, par un nœud de transformation du type `Transform2D`. Ainsi, si on déclare une route entre le champ `translation_changed` d'un nœud `PlaneSensor2D` et le champ `translation` d'un nœud `Transform2D`, le contenu graphique de ce dernier sera déplacé entre la pression et le relâchement du bouton de la souris sur les objets voisins du nœud `PlaneSensor2D`.

```
eventOut SFBool isActive FALSE
eventOut SFVec2f trackPoint_changed 0 0
eventOut SFVec2f translation_changed 0 0
```

Code 4.7 – Evènements de sortie d'un nœud BIFS PlaneSensor2D

L'approche du modèle évènementiel DOM Events est à l'opposé. Les évènements sont représentés par des structures dont les membres ne sont accessibles que par script. Les seules modifications qu'il est possible de faire sans script sont du premier type, c'est-à-dire des traitements booléens basés sur la détection d'un évènement. Ces traitements correspondent à l'action par défaut associée aux gestionnaires d'évènements. Par exemple, le nœud `a` du langage HTML peut être utilisé comme gestionnaire d'évènement avec comme action par défaut l'activation de l'hyperlien. De même, les animations dans le langage SVG ont pour action par défaut le déclenchement de cette animation.

Le Code 4.8 donne une description simplifiée de l'interface `MouseEvent` pour les évènements souris. Pour effectuer l'opération de glisser déplacer, avec ce type d'évènement, selon le modèle DOM, il faut déterminer quand le bouton de la souris est appuyé, puis sauver la valeur initiale des champs `clientX` et `clientY`, et ensuite la soustraire aux évènements de souris consécutifs jusqu'à ce que le bouton de la souris soit relâché.

```
interface MouseEvent : UIEvent {
  readonly attribute long             screenX;
  readonly attribute long             screenY;
  readonly attribute long             clientX;
  readonly attribute long             clientY;
  readonly attribute boolean          ctrlKey;
  readonly attribute boolean          shiftKey;
  readonly attribute boolean          altKey;
  readonly attribute boolean          metaKey;
  readonly attribute unsigned short   button;
  readonly attribute EventTarget      relatedTarget;
}
```

Code 4.8 – Evènement Souris selon la norme DOM Events 2

4.2.5 Résumé

En résumé, on peut donc retenir que les mécanismes sous-jacents au caractère interactif d'une scène s'appuient sur un modèle évènementiel ; que ce modèle évènementiel diffère d'un langage à l'autre en termes d'évènements traités, de type de propagation et de complexité de la description. On comprend ainsi mieux la difficulté, d'une part, de l'édition de scènes interactives et d'autre part, de la conception d'algorithmes de transcodage d'interactivité.

4.3 Scénarios d'utilisation des technologies d'interactivité

Comme nous l'avons décrit précédemment, l'interactivité implique une action de l'utilisateur, un traitement intelligent de cette action se traduisant par une suite de modifications de la scène. La valeur ajoutée à une scène par l'interactivité dépend de la quantité et de la complexité des traitements effectués. Cependant, sur certains terminaux, il n'est pas possible, avec une vitesse raisonnable, d'effectuer certains traitements complexes (calculs mathématiques, manipulation d'arbre, …). Dans une architecture client/serveur où le client possède peu de capacité de traitement, il peut être intéressant de déporter ces traitements au niveau du serveur. A l'inverse, si on sait que le client a des capacités importantes et que l'on souhaite alléger la tâche du serveur, et servir plus de clients, on peut effectuer les traitements au niveau client. Il existe donc deux types d'architectures de services interactifs : ceux où l'intelligence du service est centralisée sur un serveur et ceux où l'intelligence est déportée au niveau du client. Nous présentons, dans cette partie, le fonctionnement de ces deux types d'architecture pour en proposer ensuite une synthèse.

4.3.1 Interactivité côté client : les applications AJAX

Les services interactifs, où l'intelligence de l'application est déportée au niveau du client, sont de plus en plus répandus dans le monde Internet sous le nom de services ou applications AJAX. AJAX est un acronyme qui signifie "*Asynchronous JavaScript And XML*" [83]. Ce terme regroupe un ensemble de technologies permettant de créer des services interactifs plus réactifs pour Internet que les services traditionnels.

Le problème que tente de résoudre AJAX est la lenteur de l'interactivité sur Internet. Dans l'approche classique, si une action de l'utilisateur nécessite de modifier la scène en cours de visualisation avec des données issues du serveur, le terminal client (navigateur Web) envoie une requête HTTP avec des paramètres issus de la scène courante et de l'action utilisateur. Le serveur effectue alors le traitement et transmet une nouvelle scène.

Cette méthode pose deux problèmes : celui de l'information redondante retransmise d'une scène à l'autre, et celui du temps de traitement de la requête par le serveur. Plus la scène est volumineuse ou plus le temps de traitement est long, plus l'utilisateur attendra. Un exemple illustrant cette lenteur serait une page HTML présentant deux composants graphiques du type `ComboBox` où le contenu de la seconde dépend du choix dans la première et du contenu d'une base de données située au niveau d'un serveur potentiellement différent du serveur de l'application. Dans une architecture sans moteur AJAX, une nouvelle page HTML doit être retransmise à chaque changement dans la première `ComboBox`.

Une application AJAX comprend les technologies suivantes :

- des technologies de descriptions de scènes comme SVG ou HTML, incluant des capacités d'interaction (avec souris, clavier, …), telles que décrites précédemment;
- une technologie pour manipuler et modifier des scènes, comme les interfaces DOM ;
- une technologie pour faire des requêtes asynchrones à un serveur de données. Les données reçues sont généralement des données XML et l'API permettant les requêtes de données XML sur le protocole HTTP est appelée `XMLHttpRequest` [68];
- et enfin, des bibliothèques de code ECMAScript faisant appel aux technologies précédentes, appelé le moteur AJAX.

La Figure 4.7 décrit l'imbrication de ces technologies dans une application AJAX.

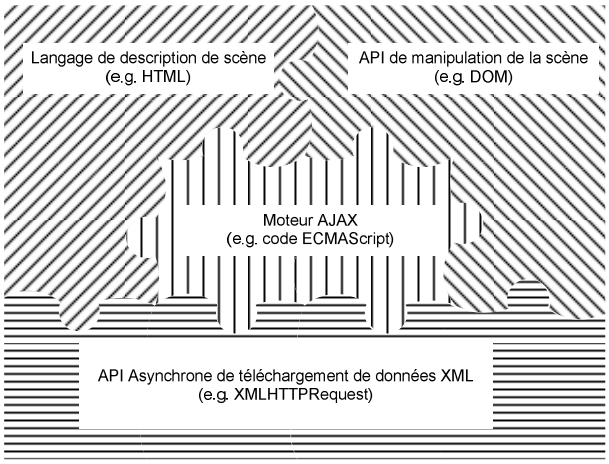


Figure 4.7 – Architecture d'une application AJAX

Le principe d'une architecture AJAX est d'introduire le moteur de traitement de l'interactivité dans la scène. Ce moteur AJAX grâce à l'objet `XMLHttpRequest` est capable de faire des requêtes asynchrones auprès de différents serveurs et, grâce à l'interface DOM, de transformer la réponse XML en appels à des fonctions de manipulation de la scène en cours. Cette approche présente plusieurs avantages :

- Elle permet de réduire la quantité d'information redondante transmise par le serveur Web : le serveur ne transmet que les nouvelles données.
- Elle permet de décorréler le service de données de l'utilisation qui en est faite. Le serveur de données fournit des documents XML que le moteur AJAX transforme ensuite selon les besoins particuliers de l'application AJAX.
- Enfin, le traitement asynchrone permet d'utiliser le temps entre l'envoi de la requête et la réception des données pour faire d'autres traitements, voire pour laisser la main à l'utilisateur et améliorant ainsi la réactivité de la scène.

Bien sûr, ce modèle présente deux inconvénients liés au fait qu'il déporte l'intelligence au niveau client. Le client doit faire le traitement que le serveur effectuait auparavant, ce qui peut être difficile sur des téléphones mobiles par exemple. Ensuite, la bande passante initiale peut être plus importante si le moteur AJAX est compliqué et nécessite beaucoup de code ECMAScript.

Bien que le nom AJAX soit essentiellement associé à un ensemble de technologies issues du W3C, il est tout à fait envisageable avec d'autres technologies comme BIFS ou Flash. D'ailleurs, le langage ActionScript utilisé par le langage de descriptions de scènes Flash intègre, depuis la version 6, des API pour satisfaire les besoins d'un service AJAX. Tous les composants nécessaires pour une architecture d'application AJAX sont disponibles. En effet, le lecteur Flash intègre des objets ActionScript, appelés `LoadVars` et `XML`, qui permettent respectivement d'effectuer une requête vers un serveur et de traiter le contenu du document XML reçu pour modifier la scène. Le Code 4.9 illustre comment ces opérations peuvent être réalisées dans le langage ActionScript [95].

```
document = new XML();
document.onData = function {
 var a = document.firstChild.childNodes[0].attributes.monAttribut;
 ...
}
loader = new LoadVars();
loader.sendAndLoad("http://ajaxserveur.com/service.xml", document);
```

Code 4.9 – Moteur AJAX simplifié utilisant des objets ActionScript

4.3.2 Interactivité côté serveur : les applications MPEG-4

Comme nous l'avons vu, la norme MPEG-4 BIFS définit un langage de descriptions de scènes comprenant des moyens permettant l'interactivité locale. Les nœuds tels que `TouchSensor`,

`InputSensor`, `PlaneSensor2D` … permettent de détecter des actions utilisateurs et de déclencher des actions (soit l'exécution de script, soit celle de nœuds `Conditional`, soit le déclenchement d'évènements). De plus, la capacité d'intégrer, dans une scène MPEG-4, du code ECMAScript utilisant les API VRMLScript offre la possibilité d'effectuer des manipulations complexes de l'arbre de scène BIFS.

Cependant, la norme MPEG-4 BIFS propose également des outils permettant de concevoir des services interactifs où le traitement complexe d'une interaction est effectué sur un serveur distant. Ainsi, grâce au nœud `ServerCommand`, il est possible de renvoyer des informations au serveur qui a transmis la scène initiale. Ensuite, celui-ci interprète l'interaction et produit un résultat sous la forme d'une série de modifications de la scène, représentées par des BIFS Updates. Enfin, il les transmet au client par le canal de transmission de la scène initiale. De même, grâce au nœud `Anchor`, il est également possible d'effectuer une requête vers un serveur avec des paramètres. Néanmoins, à l'expérience, on s'aperçoit rapidement que ces deux mécanismes (`ServerCommand`, `Anchor`) ont été conçus pour des services interactifs simples et dans une architecture de service à forte connotation *Broadcast*, c'est-à-dire où il n'existe qu'une seule voie descendante possible entre le serveur et le client, celle du canal *Broadcast*. Ensuite, on remarque que dans cette architecture, les données transportées sur cette voie descendante sont des données BIFS. Donc, dans tous les cas, le serveur doit connaître la scène (identifiants des nœuds, paramètres d'encodage) pour transmettre ces informations aux bons éléments de la scène. Le serveur d'envoi ne peut pas être indépendant ni du format BIFS, ni de l'architecture de la scène cible comme c'est le cas dans les architectures d'interactivité du type AJAX.

4.4 Conclusion

Dans ce chapitre, nous avons proposé une synthèse des mécanismes d'interactivité dans les scènes multimédia notamment dans le format VRML et ceux liés au W3C. Nous avons présenté les grandes méthodes pour capter et propager les évènements ainsi que pour modifier la scène. Nous avons également présentés les scénarios typiques actuels. Nous concluons cette partie en constatant qu'à ce jour, la majeure partie de l'interactivité dans les scènes multimédia est encore décrite en utilisant un langage de script, mais l'arrivée des standards comme LASeR ou REX devrait augmenter l'utilisation des mécanismes d'interactivité par mise à jour et développer les services interactifs centralisés.

Chapitre 5 Descriptions de scènes, création et distribution

5.1 Introduction

Afin de comprendre les choix qui peuvent être faits lors de la sélection (ou la standardisation) d'outils pour la spécification d'un langage de descriptions de scènes (ou d'une norme), il est important de présenter les différents traitements applicables aux descriptions de scènes dans une chaîne de distribution de contenus multimédia. En effet, les choix faits dans un langage de descriptions de scènes peuvent avoir un effet positif à un endroit de la chaîne mais un effet négatif ailleurs. Par exemple, un outil peut être conçu pour être simple d'utilisation et faciliter la tâche de création de contenu mais s'avérer très difficile à gérer lors de la phase de lecture. A l'inverse, les tâches qui sont simples à traiter au niveau du client peuvent être trop détaillées pour que le créateur de contenu les utilise.

Dans ce chapitre, nous envisageons une chaîne de distribution de contenu multimédia comportant trois phases : la phase de création des contenus, la phase de diffusion du contenu sur les réseaux de télécommunication et la phase de consommation du contenu sur le terminal récepteur. Nous donnons un rapide état de l'art des techniques de création et de distribution de contenus multimédia. Les problèmes liés à la phase de consommation seront décrits dans le chapitre 7.

5.2 Edition de scènes

Le cycle de vie d'un contenu multimédia débute avec la phase d'édition. Cette phase implique l'édition des médias et celle de la scène. De nombreux outils d'édition existent pour créer les média qui composent une scène. Différents outils existent également pour créer la scène, c'est-à-dire pour concevoir l'organisation spatiale, temporelle et interactive du contenu. Un état de l'art détaillé des outils et des paradigmes de création de scène peut être trouvé dans la thèse de S. Boughoufalah [31].

Sur la base de ce travail, nous considérerons qu'il existe deux méthodes pour produire des scènes multimédia :

La première méthode est la création utilisant un outil avec une interface graphique (GUI), comme l'outil Flash. C'est le cas de la majeure partie des processus de création contenus créés professionnellement. Dans ce cas, l'utilisateur ne voit pas les possibilités offertes par le format de représentation de scène qu'il utilise. Les limitations auxquelles il se heurte peuvent être dues au langage de descriptions de scènes utilisé, mais sont souvent le fait des limitations de l'outil de création. Comme le décrit S. Boughoufalah, la génération du contenu dans ce type d'outil est souvent basée sur des modèles (*templates*) issus d'un processus de création humaine. Ces modèles sont instanciés avec un nombre restreint de paramètres, d'où les limitations. Similairement, beaucoup de contenus produits automatiquement, issus de conversion de données entre formats de descriptions de scènes, utilisent aussi les *templates*.

Dans ces outils, pour palier aux limitations du langage utilisé lors de la lecture du contenu, il arrive que la représentation de la scène interne à l'outil soit différente de la représentation finale. Dans ce cas, une étape de publication ou d'export est nécessaire. Par exemple, les produits Adobe (anciennement Macromedia) utilisent deux formats différents : un format d'édition (fichiers « .fla ») et un format de publication (fichier « .swf »).

La seconde méthode, qui reste faiblement utilisée par rapport à la première méthode, est la création de contenu à partir d'un éditeur de texte. Le créateur de contenu produit le contenu directement dans le langage final. En fonction de sa connaissance du langage, le créateur peut exploiter toutes les possibilités du langage. Ce scénario de création impose des contraintes sur le format de descriptions de la scène souvent fortes : la concision du langage, la simplicité des structures et la possibilité de réutiliser du code.

Dans les deux cas, création avec interface graphique ou création avec un éditeur de texte, la création de scènes multimédia doit prendre en compte tous les aspects d'une scène, abordés au chapitre 1, à savoir l'organisation temporelle, l'organisation spatiale, l'animation, l'interactivité et l'utilisation de média.

Les outils graphiques d'éditions sont souvent plus adaptés pour éditer l'organisation spatiale des éléments visuels de la scène. De nombreux outils existent pour tracer des primitives graphiques (vectoriel ou non), pour les agencer en 2D ou en 3D. On peut citer respectivement les outils *Adobe Illustrator* [93] ou *3D Studio Max* [97].

De même, comme le souligne S. Boughoufalah, des paradigmes graphiques existent pour faciliter l'édition temporelle (animations, synchronisation). De nombreux outils proposent à l'utilisateur de gérer des lignes temporelles sur lesquelles il place des points de synchronisation (démarrage de média, arrêt d'animation …). On peut citer les outils comme *Adobe Première* [98].

La difficulté du processus d'édition de scènes réside principalement dans l'édition de l'interactivité. En effet, comme nous l'avons décrit dans le chapitre 4, l'interactivité est dépendante du terminal cible (avec ou sans clavier, avec ou sans souris…), ainsi que du modèle évènementiel sous-jacent. Enfin, les mécanismes d'interactivité avancés sont souvent exprimés par des langages de script (JavaScript, ActionScript, …) voire des langages de programmation comme Java. Fournir un outil d'édition qui permette de créer et de déboguer des scripts est une tâche très difficile. De plus, les contenus les plus riches et les plus interactifs utilisent au maximum les mécanismes de propagation des évènements pour qu'un évènement utilisateur déclenche plusieurs actions. Il est également difficile de représenter ces phénomènes de propagation dans un outil d'édition.

Pour toutes ces raisons, beaucoup d'outils auteur limitent les possibilités d'interaction en fournissant des comportements préétablis, stockés dans des librairies d'actions. C'est par exemple le cas pour l'outil Flash qui propose une bibliothèque appelée : Composants ActionScript. Dans les versions récentes de l'outil Flash, on trouve, par exemple, dans la bibliothèque un composant "Lecteur vidéo" qui permet de contrôler interactivement la lecture d'une vidéo avec des actions du type 'lecture', 'pause', 'avance rapide'.

En conséquence, même si tout un chacun peut produire des descriptions de scènes avec un éditeur de texte, la majeure partie des contenus multimédia existants sont donc issus d'outils de création qui produisent des contenus basés sur des blocs de base graphiques et des blocs de base d'interactivité. Ces contenus sont donc : fortement redondants car ces blocs se répètent dans le contenu, et non optimisés car ces blocs sont en général conçus pour être très génériques et très détaillés. Il est donc nécessaire d'effectuer un travail d'une part sur les algorithmes de compression et d'autre part sur les algorithmes permettant d'améliorer l'efficacité de lecteurs multimédia sur ces types de contenu. Nous détaillerons nos propositions dans ce domaine dans les chapitres 6 et 7.

5.3 Diffusion de descriptions de scènes

Avec l'explosion des capacités des réseaux de télécommunication, les contenus multimédia autrefois cantonnés aux méthodes de diffusion utilisant des supports physiques (DVD), sont également diffusés sur les réseaux. Les descriptions de scènes multimédia, fortement liées aux données audiovisuelles qu'elles utilisent, sont amenées à être distribuées sur les réseaux de télécommunication selon les mêmes scénarios que ces données audiovisuelles. Parmi eux, on trouve le *streaming*, le téléchargement, et bien d'autres. Cependant, certaines particularités des descriptions de scènes impliquent soit des modifications aux mécanismes existants soit la création de nouveaux mécanismes pour que la diffusion ait lieu de manière adéquate. Dans le restant de ce chapitre, nous présentons les mécanismes de diffusions et les protocoles classiques de transport de contenus multimédia, afin, ensuite, de décrire les spécificités de la distribution de scènes multimédia.

5.3.1 Les mécanismes et protocoles de transports de contenu multimédia existants

Afin de comprendre les méthodes de distribution de contenu multimédia, nous nous intéressons dans un premier temps aux modes de distribution de contenu audiovisuel actuels. Il existe de nombreux protocoles et mécanismes de transports de données audiovisuelles. Les mécanismes actuels se caractérisent par le fait qu'ils sont adaptés pour le transport de données souvent volumineuses, à débit élevé et à fortes contraintes temporelles, comme c'est le cas pour les données audio-visuelles. Bien sûr, les données multimédia peuvent également être statiques, c'est le cas des images fixes, et ces mécanismes sont également utilisés. Les mécanismes de transport actuels peuvent être classés en trois catégories :

- le transport de fichier en mode non progressif;
- le transport de fichier en mode progressif;
- et le transport de flux (*streaming* ou *broadcasting*)

Nous faisons ici la distinction entre le téléchargement de fichier en mode non progressif ou progressif. Dans le premier cas, l'utilisateur doit attendre d'avoir téléchargé le fichier en entier avant de le visualiser. Dans le second cas, le fichier peut être vu en cours de téléchargement. Dans le cas des contenus multimédia, cette remarque est importante car la durée de téléchargement peut être longue et, dans le mode non progressif, gênante pour l'utilisateur. Nous faisons également la distinction entre téléchargement de fichier en mode progressif et *streaming* de flux car le second implique un contrôle des temps d'envoi (de réception, de décodage, ou de présentation) de morceaux identifiés du flux et une gestion du tampon de réception du client.

5.3.1.1 Les fichiers multimédia

Il existe deux catégories de fichiers multimédia : ceux qui permettent de stocker des données audio-visuelles principalement, et ceux qui permettent de stocker principalement des informations synthétiques comme des objets graphiques 2D ou 3D, des sous-titres et ne sont pas adaptés pour contenir de données audio-visuelles.

Parmi les fichiers audio-visuels traditionnels, on peut citer les formats AVI, MOV, MP4, MPG, Matroska [100], OGG [87], … Ces fichiers sont des multiplexes de données temporelles, organisés de manière à fournir un accès rapide aux éléments des flux décodables individuellement. Ils permettent de signaler les informations de synchronisation entre les données temporelles afin que le lecteur de fichier puisse présenter les flux de manière synchronisée. Enfin, certains offrent la possibilité de stocker des informations non temporelles. On peut citer le stockage de données ID3 [101] dans les

fichiers MP3, ou le stockage de données XML dans les fichiers ISO [44] par l'intermédiaire de la construction 'meta'.

L'autre catégorie de fichiers est celle qui permet de décrire uniquement les données annexes aux médias issus du monde analogique, mais ne permet pas (ou n'est pas adaptée pour) le stockage des données audio/vidéo. Ces fichiers sont le plus souvent des fichiers textuels. On peut citer les fichiers VRML (scènes 3D), SRT (sous-titres). Parmi ces fichiers textes, une part croissante correspond à des fichiers XML : XHTML, SVG, XMT, X3D, ….

5.3.1.2 Diffusion de fichiers multimédia

La diffusion de fichiers permet, comme son nom l'indique, et à l'inverse de la diffusion de flux, de disposer d'un fichier après téléchargement. Il existe de nombreux protocoles de diffusions de fichiers : SMTP, FTP, JXTA, MMS … mais seuls certains sont adaptés et utilisés dans le domaine du multimédia.

Dans la catégorie des protocoles utilisés pour le téléchargement non progressif, dans le cadre de l'utilisation de protocoles de diffusion de masse comme MPEG-2 TS ou RTP en mode *multicast*, de nombreux protocoles émergent pour la diffusion de fichiers. On peut citer les protocoles comme DSM-CC Data Carrousel ou FLUTE, qui permettent de gérer l'envoi de mise à jour du fichier et la signalisation des versions de fichiers.

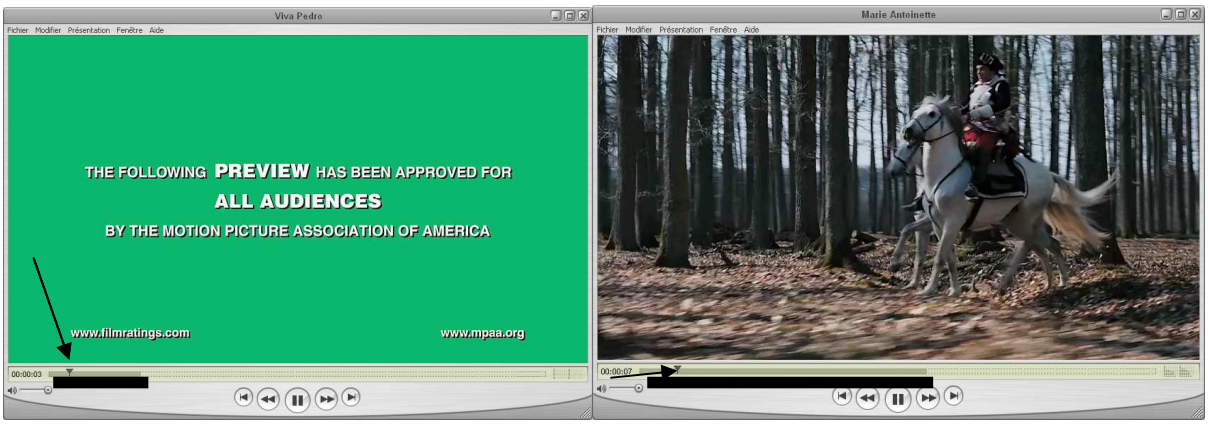


Figure 5.1 – Illustration du téléchargement progressif de vidéo

Dans l'autre catégorie, celle des protocoles permettant le téléchargement progressif, bien que de nombreux travaux se poursuivent afin de permettre le téléchargement progressif utilisant des protocoles de Peer-to-peer [35], le protocole le plus utilisé reste encore HTTP. HTTP a été initialement conçu comme un protocole de transfert de fichiers, dédié aux données textuelles, puis également aux images. HTTP offre la possibilité de faire du téléchargement progressif. On le voit par exemple lorsque le débit est faible que les pages Internet ou les images se chargent progressivement. Lorsqu'il est utilisé en conjonction avec une organisation appropriée des données dans le fichier

multimédia, HTTP permet également de transférer des fichiers audio-visuels de manière progressive. Une utilisation importante de ce mécanisme est faite pour la diffusion de radio par Internet. Un fichier MP3 virtuel est diffusé sur HTTP, mais sa taille n'est pas indiquée dans le protocole au moment de la connexion, le serveur peut alors envoyer sans cesse des données. Une autre application illustrée dans la Figure 5.1 est la visualisation de bandes annonces de films (ici issus du site Apple). Les différents lecteurs audiovisuels, comme ici QuickTime [102], illustrent la fraction de contenu en cours de visualisation (le triangle noir) et celle en cours de téléchargement (la partie grisée de la barre de progression).

5.3.1.3 Diffusion de flux multimédia

Pour ce qui est de la diffusion de données multimédia par flux, il existe deux grands protocoles : RTP, utilisé pour la diffusion sur réseau IP, en point à point ou en *multicast* ; et MPEG-2 TS utilisé sur réseau DVB, pour la diffusion de masse, traditionnellement sans voie de retour. Une des caractéristiques principales qui diffèrent entre les protocoles de distributions de fichiers et de flux, est le fait que la diffusion de flux tire profit des estampilles temporelles associées à chaque élément du ou des flux pour maîtriser finement la bande passante utilisée et éviter la congestion du réseau. Le téléchargement progressif de fichiers à l'inverse pourra consommer toute la bande passante disponible pour transmettre le fichier le plus rapidement possible.

Une autre différence réside dans le fait que les protocoles de diffusion de fichiers accordent souvent plus d'importance à l'intégrité des données plutôt qu'au caractère temps réel de la diffusion. Le choix inverse est fait dans la diffusion de flux généralement. Si un flux est distribué sur un réseau qui introduit des pertes ou des erreurs, les protocoles privilégient en général l'aspect temps-réel. Cependant, des techniques de renvois, de codes correcteurs d'erreurs, d'entrelacement de données ou de codage robuste sont également utilisées pour minimiser ces erreurs sans gêner l'aspect temps-réel.

Comme le démontrent les nombreuses activités de standardisation, les langages de descriptions de scènes tendent à être utilisés dans des environnements hétérogènes (télévision, Internet, téléphonie mobile) avec des méthodes de distribution également différentes (téléchargement de MMS, *streaming*, diffusion DVB-T/DVB-H). Bien qu'un langage de descriptions de scènes doive être indépendant du mode de distribution, il doit néanmoins être utilisable dans tous les modes. Mais la possibilité de diffuser des descriptions de scènes sur tout type de réseau n'est pas une chose simple, comme nous le décrivons ci-dessous.

5.3.2 Spécificité des descriptions de scènes

La diffusion de descriptions de scènes pose des problèmes spécifiques. Premièrement, les données de scènes sont des informations en général peu redondantes et très sensibles aux pertes. Ensuite, étant donné que certaines parties de la scène sont utilisées pour composer les éléments média, il faut que ces

parties arrivent au terminal client avant toute autre information, avant les données média notamment. Enfin, les descriptions de scènes peuvent être statiques ou temporelles, représentées sous forme textuelle ou binaire, sous forme de flux ou de document.

5.3.2.1 Spécificité des flux de scènes

Les flux de descriptions de scènes, comme nous l'avons présenté dans le chapitre 1, sont très similaires aux flux audio vidéo. Ils sont découpés en unités de base et doivent être transmis avec les mêmes contraintes temps-réel que les données média. Les protocoles classiques tels que RTP ou MPEG-2 TS peuvent être utilisés sans grande difficulté si quelques particularités des descriptions de scènes sont respectées. Nous présentons ici ces particularités en nous appuyant sur une analogie avec le codage vidéo.

5.3.2.1.1 Analogie avec les flux vidéo

Dans le domaine du codage vidéo, on représente traditionnellement une image dans le domaine non compressé par un tableau de pixels, alors que dans le domaine compressé, certains codages comme MPEG-2, utilisent deux types d'images : des images dites intra (I) et des images de prédictions (P ou B), qui décrivent des modifications par rapport à une ou plusieurs images de référence.

Pour comprendre les difficultés de la diffusion des flux de scènes, on peut faire l'analogie entre un flux vidéo et un flux de descriptions de scènes. Conceptuellement, un flux dans le domaine non compressé peut être vu comme la succession de descriptions complètes de la scène à différents instants consécutifs. L'étape de compression consisterait à calculer les différences entre deux descriptions de scènes. Ainsi dans le domaine compressé (binaire ou textuel), le flux de descriptions de scènes comporte deux types de données : des descriptions complètes de la scène (équivalentes aux images I) et des mises à jour de cette scène (équivalentes aux images P).

Cette analogie entre le flux de descriptions de scènes et le flux vidéo est intéressante car elle permet de comprendre que l'on peut appliquer au flux de description de scène toute sorte de principes souvent appliqués à la vidéo. Parmi ces principes, les politiques de codage facilitant l'accès aléatoire, et celles améliorant la résistance aux problèmes de transmissions sont intéressantes.

5.3.2.1.1.1 Points d'accès aléatoire

Dans le domaine du codage vidéo, on peut coder une séquence longue avec une seule image I suivie d'images P. L'avantage de cette méthode est le faible débit nécessaire pour coder la séquence du fait de l'utilisation de prédiction temporelle. L'inconvénient en revanche est la forte sensibilité aux erreurs de transmission. Pour cette raison, on utilise régulièrement des images I au milieu du flux pour permettre une resynchronisation en cas de perte ou de corruption des paquets lors de la transmission, ou permettre à l'utilisateur de joindre une diffusion en cours. Le même principe est applicable aux flux de descriptions de scènes. On introduit dans le flux une nouvelle scène intermédiaire complète, résultat

de l'agrégation de la scène initiale ou de la scène intermédiaire précédente et des mises à jour consécutives.

Les mêmes compromis se posent pour le codage des flux de descriptions de scènes que pour le codage des flux vidéo. Il est évident que des remplacements entiers de la scène sont plus coûteux à représenter et à coder que la mise à jour correspondante. La Figure 5.2 illustre ce compromis. La courbe pleine épaisse montre le débit d'un flux de descriptions de scènes codé sans remplacement de scène intermédiaire, avec un débit moyen de 251 kb/s. La courbe en pointillés serrés montre le débit de la même scène encodée avec des remplacements de scènes toutes les 3 secondes (c'est-à-dire toutes les 36 trames), pour un débit moyen de 284 kb/s, soit 14.5 % d'augmentation de débit. La courbe en pointillés fins montre la même scène encodée avec des remplacements toutes les 5 secondes (toutes les 60 trames), pour un débit moyen de 272 kb/s, soit 8.5% d'augmentation de débit.

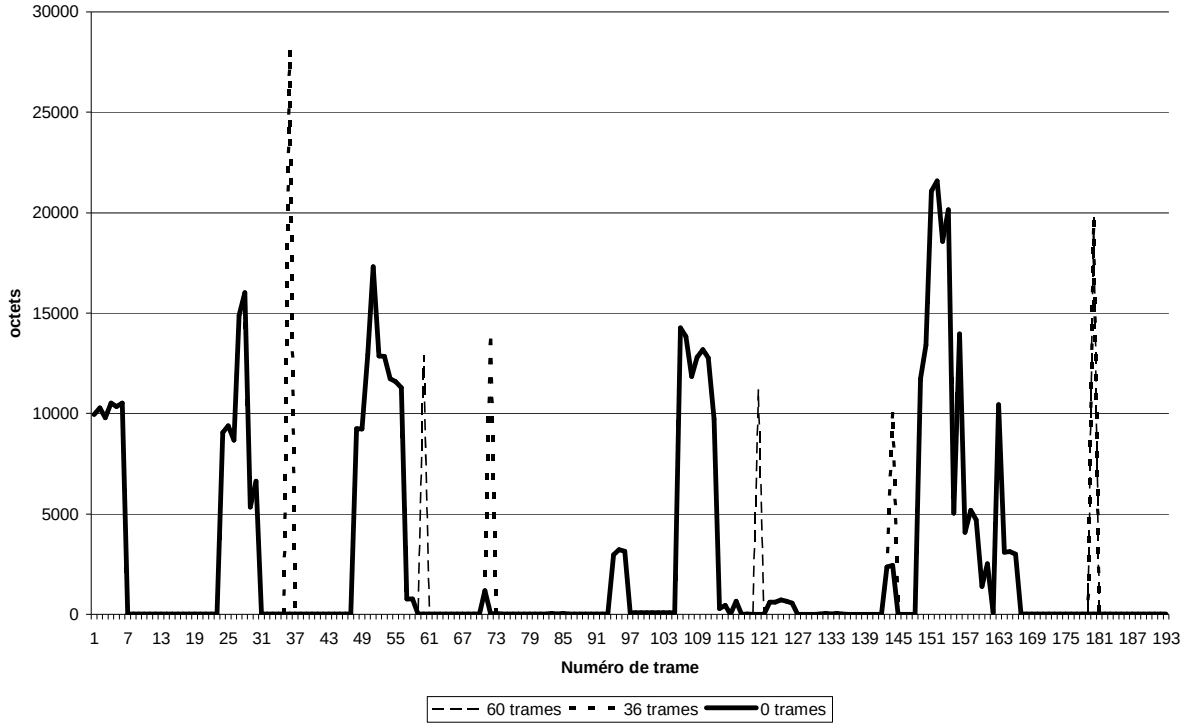


Figure 5.2 – Débits de flux de description d'une scène selon la période des remplacements de la scène

Les images I intermédiaires en vidéo servent également comme point d'accès aléatoire (RAP, *Random Access Point*) dans le contenu. Elles évitent ainsi d'avoir à décoder l'image I initiale puis toutes les images P pour accéder au milieu de la séquence. Les trames de remplacement de scène jouent également ce rôle, avec une différence dans le cas de scènes interactives.

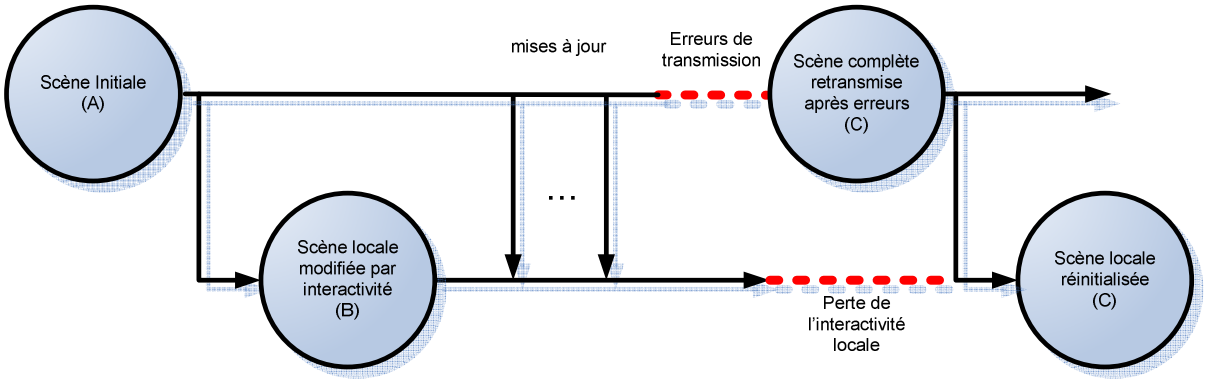


Figure 5.3 – Impact des erreurs de transmission sur l'interactivité

Imaginons, par exemple, le cas d'une scène interactive mise à jour par un serveur distant décrite par la Figure 5.3. Supposons que le client ait interagit localement avec le contenu et qu'il visualise donc la scène dans un état B différent de l'état initial A. Si des pertes sont subies dans l'envoi des mises à jour de la scène, le client peut utiliser un point d'accès aléatoire. Cependant, ce dernier est envoyé par le serveur. Or, le serveur n'a pas connaissance des interactivités locales car de nombreux clients sont connectés en parallèle. Par conséquent, la scène RAP envoyée (état C) correspond à l'état A auquel on a appliqué les mises à jour envoyées depuis l'état A. Dans ce cas, la resynchronisation avec le flux permettra au client de suivre la suite des mises à jour mais lui fera quitter l'état B. Une des solutions serait de sauvegarder localement dans une variable le fait que le client soit dans l'état B et de recharger cette variable après toute resynchronisation. A l'heure actuelle, seul le standard LASeR permet ce comportement, grâce aux commandes `Save` et `Restore`.

Enfin, un autre problème lié à ces envois de point d'accès aléatoire est la gestion du temps et notamment la reconstruction du temps de scène, aspect décrit au chapitre 1.

5.3.2.1.1.2 Allocation de débit

L'avantage de la structuration d'une scène sous forme de flux est de pouvoir diffuser en continu les flux de descriptions de scènes comme les flux audio-visuels. Mais, le fait qu'une unité de base d'un flux de scène soit découpée sous forme de mises à jour potentiellement indépendantes des autres, et notamment de la précédente, rend également possible le lissage du débit nécessaire pour transmettre une séquence. Pour continuer l'analogie avec la vidéo, il est possible de mettre en œuvre des politiques d'allocation de débit. Il est intéressant de profiter des périodes où les mises à jour de la scène sont peu nombreuses ou peu coûteuses pour transmettre en avance des éléments qui ne seront utilisés que plus tard dans la scène, tout en respectant un débit maximal. Cependant, pour être efficace, l'utilisation de telles techniques doit prendre en compte l'occupation mémoire maximale au niveau du client.

La Figure 5.4 illustre le résultat du lissage de débit d'un flux de descriptions de scènes. La courbe en pointillé montre le débit non lissé alors que la courbe pleine montre le débit lissé pour un seuil maximal de 64 kb/s.

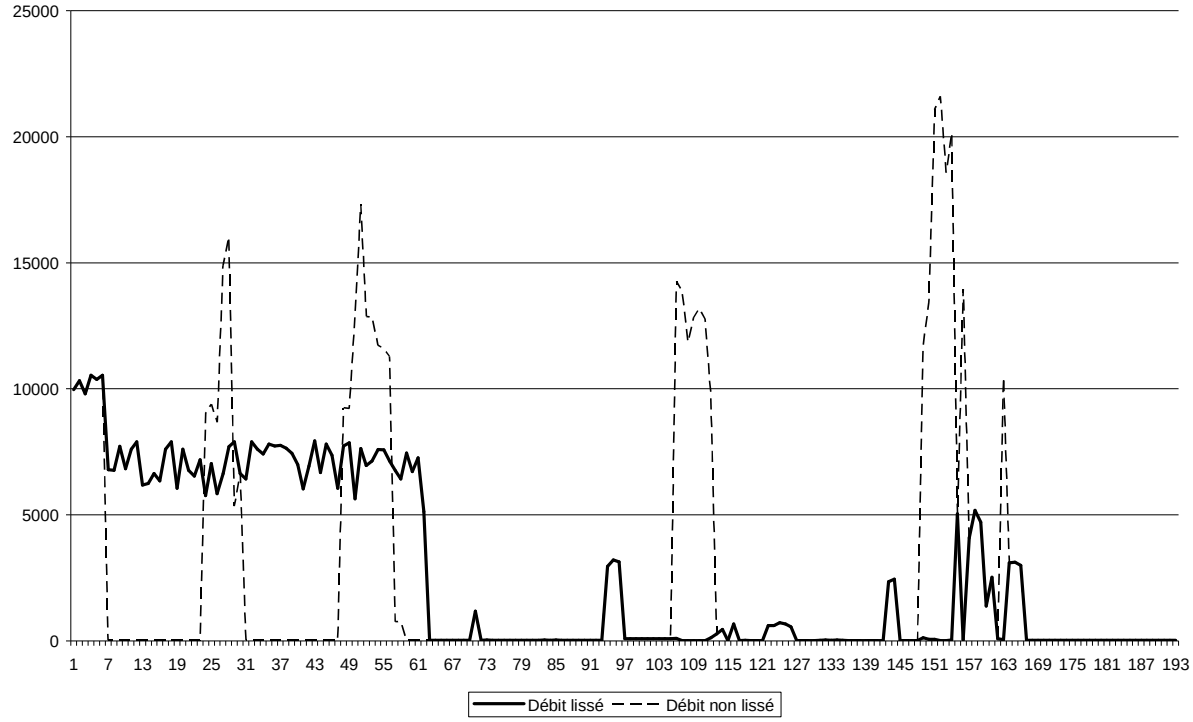


Figure 5.4 – Débit initial et débit lissé (en octets) d'un flux de descriptions de scènes en fonction du numéro de trame

5.3.2.1.2 Adaptation de flux de scènes

L'hétérogénéité des réseaux, des terminaux et des contextes d'utilisation contraignent de plus en plus les distributeurs de contenus à envisager des méthodes de distribution intégrant des modules d'adaptation permettant, suite à une négociation client-serveur, d'adapter le contenu au contexte d'utilisation. Différentes techniques existent pour adapter les média audiovisuels. Nous présentons ici les mécanismes génériques qui permettent d'adapter les média scalables. Nous verrons dans le chapitre 6 comment ces mécanismes peuvent être déclinés également pour l'adaptation de flux de description de scènes multimédia.

5.3.2.1.2.1 Les notions de scalabilité

Il existe différentes définitions et interprétations de la notion de scalabilité. Dans le vocabulaire MPEG-4, un média est scalable s'il est organisé en N flux : un flux représentant la couche de base et les autres flux représentant des couches d'améliorations. Ces flux sont décrits selon la norme MPEG-4 Système par des descripteurs (`ElementaryStreamDescriptor`) eux-mêmes contenus dans des descripteurs d'objets média (`ObjectDescriptor`). La présence d'une dépendance d'un flux

d'amélioration vers un autre flux d'amélioration indique qu'il s'agit d'un ensemble de flux média représentant un objet média scalable.

La notion de scalabilité en MPEG-21 est légèrement différente. Elle ne considère pas un ensemble de flux mais un seul flux. La norme MPEG-21 définit un flux scalable comme :

« Un flux binaire dont les données sont organisées de telle manière que lors de l'obtention de ce flux, il est possible de visualiser une version dégradée de la ressource et ensuite d'améliorer cette visualisation en chargeant des données supplémentaires. »

Or comme nous l'avons vu précédemment, une unité d'accès (AU) BIFS est composée de commandes. Parmi toutes les commandes disponibles, la commande `Insert` remplit les fonctions nécessaires à ce comportement de scalabilité. Si on considère un flux BIFS (Figure 5.5), où les unités d'accès sont organisées de telle sorte que la fin de l'unité d'accès soit composée de commandes d'insertion, éventuellement de remplacement, on peut dire qu'un tel flux BIFS est un flux scalable au sens MPEG-21. De même, un flux LASeR et un flux Flash, grâce aux commandes d'ajout d'objets dans le dictionnaire, peuvent être vus comme des flux scalables au sens MPEG-21.

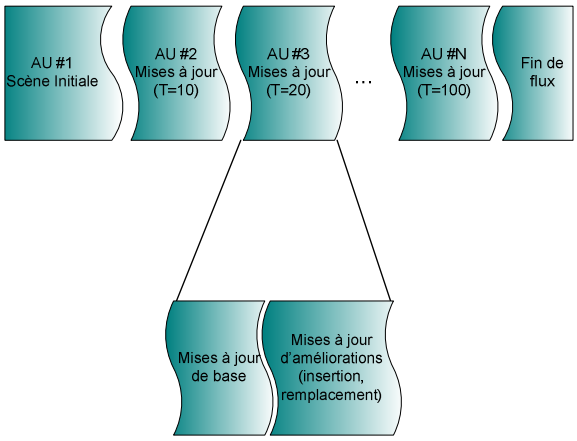


Figure 5.5 – Structure d'une unité d'accès de flux de scène scalable

Note : Le terme scalabilité est également utilisé, notamment en SVG, pour désigner une caractéristique des images vectorielles : le fait qu'elles puissent être zoomées sans perte de qualité. Il s'agit d'une notion différente mais on peut dire que BIFS, LASeR et Flash sont des langages scalables à double titre.

5.3.2.1.2.2 Principes de l'adaptation de flux scalables selon la norme MPEG-21

L'adaptation est traitée par le groupe MPEG dans la norme MPEG-21, plus particulièrement dans la partie MPEG-21 *Digital Item Adaptation* (DIA) [36]. Celle-ci définit la notion d'adaptation de flux scalable comme le processus qui consiste à transformer ou adapter un flux scalable pour satisfaire certaines contraintes utilisateurs, réseaux, environnementales… Le processus d'adaptation d'un flux scalable est décrit dans la Figure 5.6, tirée de la norme MPEG-21 DIA.

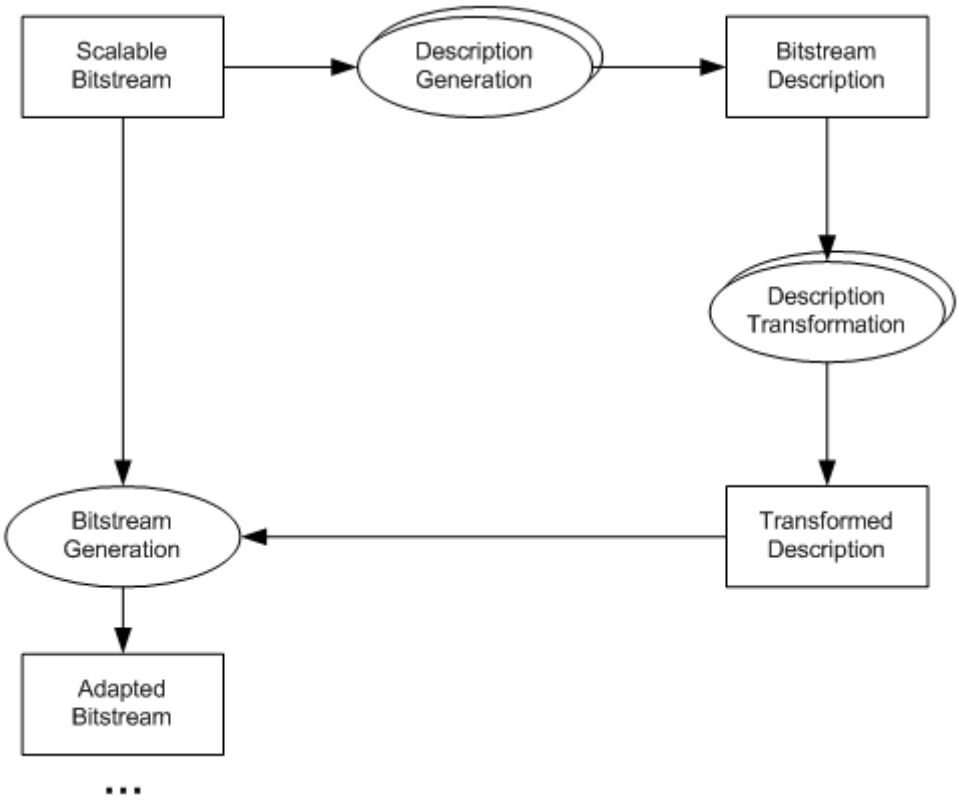


Figure 5.6 – Processus d'adaptation de flux scalable selon la norme MPEG-21

Ce processus consiste à produire une description XML, au format gBSD [41] ou BSD [37], du flux scalable. Cette description peut être produite au moment de l'encodage ou juste avant la diffusion, par un module connaissant le format binaire. Une description gBSD décrit la structure d'un flux en indiquant, pour chaque élément de la syntaxe binaire[7], sa position dans le flux et sa longueur. Un marqueur est également associé à chaque élément de syntaxe. Ces informations sont regroupées dans un élément XML nommé `gBSDUnit`. Cette description est ensuite adaptée par une transformation de document XML du type XSLT [64] ou STX [78] pour produire une nouvelle description gBSD. Cette nouvelle description correspond à un flux adapté théorique. Elle est produite en supprimant (ou en modifiant) des éléments de la description gBSD initiale sur la base des marqueurs. Par exemple, si un marqueur indique qu'un élément de syntaxe du flux appartient à une couche d'amélioration non désirée, cet élément n'est pas copié dans la description gBSD adaptée. Le flux média est ensuite produit à partir du flux d'origine et de cette description gBSD adaptée.

[7] La granularité de la description du flux binaire est laissée libre.

L'avantage de cette technique est que la transformation du document et la génération du flux adapté sont deux opérations indépendantes du format du flux média. Ces traitements peuvent être effectués par des modules génériques. L'inconvénient est le surcoût de débit introduit par la description gBSD. Cependant, il existe des techniques de compression efficaces sur ce type de documents [27].

5.3.2.2 Spécificité des documents de scènes

Dans la partie précédente (5.3.2.1), nous avons présenté les méthodes existantes pour transporter et adapter les flux de descriptions de scènes. Dans cette partie, nous exposons les difficultés de transport des scènes multimédia quand la description de la scène est représentée sous forme de document et non de flux, et surtout que ce document est volumineux.

Souvent, un document XML nécessite d'être reçu complètement avant de pouvoir être interprété, mais dans le cas des documents XML représentant des descriptions de scènes ce n'est pas toujours le cas. En effet, on le voit quand on télécharge un fichier HTML, la page s'affiche généralement avant que le fichier entier n'ait été reçu. Le langage SVG exploite également la notion de visualisation progressive de document. La Figure 5.7 montre le résultat du chargement et de la visualisation progressifs d'un fichier représentant un contenu graphique vectoriel. Le contenu est joué dans le lecteur SVG que nous avons développé au cours de cette thèse, présenté dans le chapitre 7. Les images successives à 21%, 36%, 50% et 100% du chargement donnent l'impression que le contenu se construit progressivement.

Cependant, le langage SVG introduit en plus la possibilité de contrôler finement la visualisation progressive. En effet, par défaut, un sous arbre de l'arbre de scène SVG peut être rendu avant d'avoir été complètement téléchargé mais l'attribut `externalResourcesRequired` permet de signaler au niveau d'un sous arbre que la visualisation de ce dernier ne peut arriver avant l'évènement de chargement `load` correspondant à ce sous arbre. Un exemple de code est fourni dans le Code 5.1. Le résultat de l'affichage de ce fragment de document SVG sera vide du fait de l'attribut `externalResourcesRequired` tant que la balise fermant l'élément g (`</g>`) ne sera pas reçue.

```
<svg xmlns="http://www.w3.org/2000/svg" width="450" height="450">
 <g externalResourceRequired="true">
  <circle cx="50" cy="200" r="35" fill="red" stroke="black" />
  <circle cx="150" cy="200" r="35" fill="orange" stroke="black" />
```

Code 5.1 – Exemple de fragment de document SVG utilisant l'attribut externalResourcesRequired

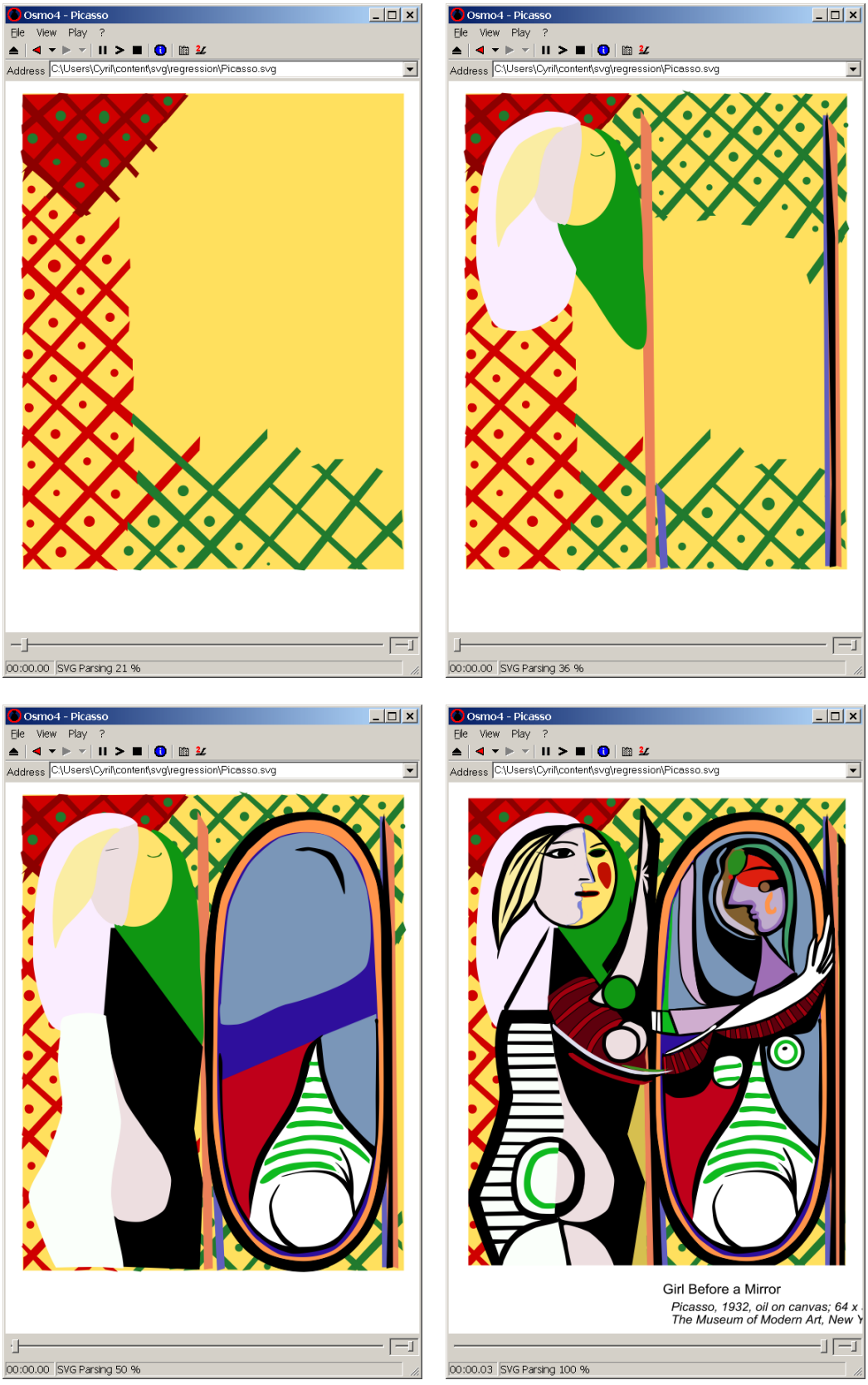


Figure 5.7 – Illustration du chargement progressif d'un contenu graphique vectoriel

Ainsi, dans le langage SVG, on peut afficher des éléments blocs par blocs. Cette technique est très avantageuse mais possède un inconvénient. Comme l'ordre d'affichage dans un document SVG est déterminé par l'ordre de déclaration dans le document, les blocs affichés en premier seront les blocs correspondant au fond, ce qui peut être ennuyeux car dans la plupart des scènes, le contenu important est placé au premier plan, donc en dernier dans le document. En prenant en compte cette particularité, on peut dans certains cas, construire les contenus de sorte que les informations importantes soient mises au début du document, mais ce n'est pas systématiquement possible.

5.4 Conclusion

Dans ce chapitre, nous avons présenté deux aspects de la chaîne de distribution de descriptions de scènes multimédia : la partie création et la partie distribution. Nous avons exposé brièvement l'état de l'art des méthodes d'édition de descriptions de scènes pour souligner le fait que les descriptions de scènes crées par des outils d'édition automatique nécessitaient des optimisations pour obtenir une représentation des scènes plus compacte à transmettre et moins complexe à lire. Nous avons également présenté l'état de l'art des différents modes de distribution des contenus multimédia, pour souligner les spécificités des descriptions de scènes lorsqu'il s'agit de leur diffusion. Nous avons notamment présenté les problèmes de la diffusion de document de scène qui pour l'instant est limité au mode téléchargement progressif. Nous avons également introduit les concepts de scalabilité et d'adaptation de flux multimédia. Dans les chapitres suivants, nous proposons des méthodes pour résoudre ces différents problèmes, notamment pour les scènes animées.

Chapitre 6 Méthodes pour la représentation efficace de scènes multimédia animées

6.1 Introduction

Dans les chapitres précédents, nous avons présenté notre analyse de l'état de l'art des langages de descriptions de scènes. Nous avons décrit les principaux formats existants et présenté une synthèse des mécanismes d'animation de scènes, des méthodes de compression de scènes, des méthodes d'interactivité et des méthodes d'édition et de distribution de scènes.

Le constat que nous faisons à la suite de cette analyse est qu'il n'existe à l'heure actuelle aucune méthode pour représenter une scène multimédia animée qui satisfasse les critères suivants : qui soit à la fois efficace en débit, qui offre des propriétés de *streaming*, de scalabilité, et qui permette une maîtrise de la consommation mémoire et de la charge de calcul au niveau du lecteur. Dans ce chapitre, nous proposons un ensemble de méthodes pour permettre la représentation efficace de scènes multimédia fortement animées satisfaisant ces critères. Nous nous intéressons précisément à améliorer l'efficacité des différentes méthodes existantes dans les domaines de la compression, de la diffusion et de l'adaptation. Nous présentons ces différentes améliorations au travers d'expériences effectuées avec les différents langages présentés dans le chapitre 1.

6.2 Cas d'étude : le dessin animé

Tout au long de ce chapitre, nous illustrerons nos travaux sur une catégorie de scènes où l'animation est particulièrement importante. Il s'agit des contenus graphiques vectoriels animés de type dessins animés ou bandeaux publicitaires en 2D. Cette catégorie de scènes est très répandue sur Internet et y représente pour l'instant la majeure partie des contenus synthétiques animés. Les propositions, que nous donnons dans ce chapitre, visent à permettre la consommation de ces contenus sur des terminaux autres que les PC connectés à Internet, comme par exemple les terminaux mobiles connectés à un réseau 3G.

Pour comprendre les propositions qui suivent, nous décrivons dans un premier temps les hypothèses de travail. Nous présentons brièvement les processus de création des contenus animés auxquels nous nous sommes intéressés. Nous verrons que ces processus peuvent être facilement généralisés à la création de scènes multimédia complètes et ainsi que notre méthode de représentation de scènes peut être généralisée.

Le processus de création de contenu vectoriel animé, du type dessin animé, et plus généralement de scènes multimédia animées, est un processus artistique qui implique des scénaristes et graphistes. Il s'agit d'un processus long et complexe qui a besoin d'être optimisé pour permettre la création de séquences longues, de manière efficace, surtout lorsque les scènes font appel à de nombreux objets graphiques synthétiques. De nombreux logiciels spécifiques, différents de ceux présentés dans le chapitre 5, existent pour faciliter ce processus. Ces logiciels utilisent souvent des formats de représentations internes propriétaires et ensuite permettent l'export, la publication, généralement dans un format vidéo ou dans le format Flash. Les propositions de représentation que nous décrivons dans ce chapitre ont vocation à être intégrées dans les fonctions d'export de ces outils. On distingue deux catégories d'outils qui se différentient sur le format des données entrantes : des dessins réalisés informatiquement ou des dessins sur support papier :

- Il existe de nombreux outils professionnels, et non professionnels, de la première catégorie. On peut citer le logiciel *Flash* ou le logiciel de dessin vectoriel *Illustrator* [93] combiné avec le logiciel *Mobile Designer* [94] pour effectuer les animations. Ces logiciels permettent de créer des dessins animés comme illustré dans la Figure 6.1. Le processus de création de chaque trame d'animation d'un objet graphique vectoriel est un processus long si l'on décide de créer chaque image séparément. L'utilisation des mécanismes d'animation par interpolation décrit dans le chapitre 2 permet d'accélérer ce processus en limitant le nombre d'image clés à créer. Les logiciels précédemment cités utilisent ces mécanismes d'interpolation.

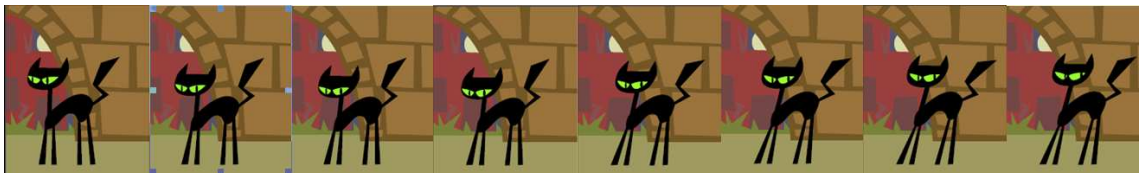


Figure 6.1 – Exemple de dessin animé paramétrique créé avec Mobile Designer

- La seconde catégorie utilise des dessins sur support papier. Ces logiciels, comme *ePegs* et *Toon Boom Studio* [92], sont les logiciels historiques du domaine et sont encore ceux qui sont utilisés le plus par l'industrie de production des dessins animés de type classique (*Tex Avery, Hanna & Barbera*), notamment pour la télévision. En effet, la qualité visuelle des dessins obtenus sur papier est jugée meilleure, et la création d'un dessin sur papier par un graphiste professionnel n'est pas nécessairement plus longue que la création informatisée. De plus, la

richesse visuelle de l'animation est souvent jugée insuffisante quand elle est produite par interpolation. Nous avons travaillé sur la base de contenus produits par cette seconde catégorie de logiciels.

Nous supposerons que le processus de création de contenu graphique vectoriel animé pour lequel nos propositions s'appliquent est le suivant :

- des dessinateurs créent les objets graphiques (personnages, décors) sur support papier ;
- les supports papiers sont acquis par scanner ;
- certaines images issues du scanner sont analysées et transformées dans un format vectoriel ;
- les dessinateurs ajoutent, à certaines images, les aplats ou gradients de couleurs aux éléments graphiques à l'aide d'un logiciel dédié ;
- les trames du scénario d'animation, appelées "prises de vues", sont créées par empilement des dessins précédents ;
- l'animation est produite par ajout et positionnement de nouveaux dessins dans une pile ou par suppression de dessins précédents ;
- et enfin, les mouvements de caméra sont ajoutés en changeant le positionnement de certains dessins.

Ce processus de création de scènes produit généralement des objets graphiques composés d'un décor sur lequel évoluent des personnages. Les dessins des personnages sont généralement composés de zones de couleur uniforme, souvent bordées d'un trait de couleur différente. Enfin, l'animation produite par ce processus se traduit par une liste d'actions à entreprendre à chaque instant : ajout d'un nouvel élément à afficher, suppression d'un élément qui était affiché dans l'image précédente, transformation d'un élément, mouvement de caméra. Les exemples de la Figure 6.2 montrent des dessins animés, ici produit par *Hanna & Barbera*, typiquement produits en suivant ce processus de création.

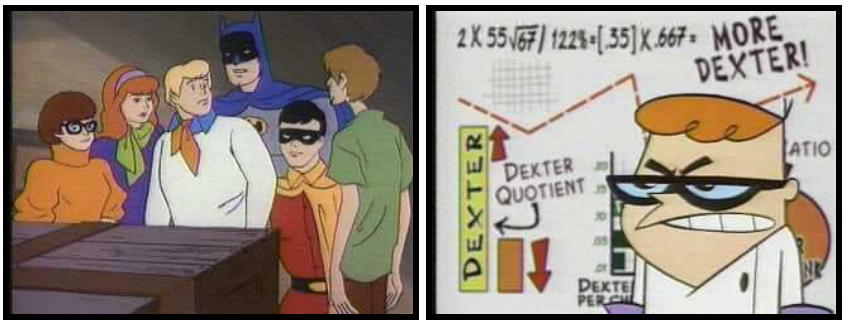


Figure 6.2 – Exemple de dessins animés classiques

Ce processus de création est intéressant car il s'étend de manière naturelle à la création de scènes multimédia animées et interactives complètes, c'est-à-dire comprenant également des éléments média de type audio/vidéo et du texte. En effet, aux dessins acquis par scanner, il suffit d'ajouter les éléments médias audio/vidéo/texte. Le placement et le positionnement des objets dans la prise de vue peuvent rester identiques pour ces nouveaux objets. Aux commandes de mise à jour de la feuille de prise de vue, on ajoute des commandes de démarrage et d'arrêt des éléments média. Enfin, une étape supplémentaire peut être facilement ajoutée à la fin de ce processus pour intégrer à la scène des capacités d'interactivité.

6.3 Choix d'une représentation

Le processus précédent aboutit donc à une liste d'objets visuels synthétiques ou non synthétiques à utiliser dans la scène, ainsi qu'à une suite d'actions à entreprendre à un instant précis pour composer une trame. Sur ces bases, nous nous sommes intéressés à déterminer la représentation de scènes la plus appropriée à la fois pour la liste des objets visuels et pour les actions d'animation.

Naturellement, la représentation de scènes utilisée par le langage Flash, utilisant les notions de dictionnaire et de liste d'affichage paraît la plus appropriée pour représenter la scène initiale. Cependant, nous avons opté pour une représentation arborescente de la scène parce que, outre le fait qu'elle permette de représenter les notions de dictionnaire et de liste d'affichage dont nous avons besoin ici, elle nous apporte plus de flexibilité dans l'organisation des objets dans la scène, ce qui peut être très utile pour la phase de création. Nous pouvons, par exemple, définir des sous-dictionnaires pour les images, les textes, et les autres catégories d'objets utilisés dans la scène.

Le processus de création des scènes animées, auxquelles nous nous intéressons, produit une liste d'actions. Le modèle d'animation par mise à jour est dans ce cas le plus approprié. Nous avons cependant introduit une hypothèse de travail supplémentaire : l’exclusion de l’utilisation de langage de script. En effet, l'exécution de scripts nécessite un interpréteur qui est coûteux en ressource mémoire et processeur au niveau du lecteur de contenu.

A partir de ces choix (représentation arborescente et animation par mises à jour sans utilisation de script), nous avons expérimenté l’utilisation des formats standards décrit précédemment (MPEG-4 BIFS, MPEG-4 LASeR, W3C SVG). Ces expérimentations ont eu pour but la réalisation de convertisseurs de contenus, à partir de scènes décrites dans le format Flash ou dans le format propriétaire de l'outil MediaPegs. Nous présentons ici les problèmes de représentation de ces scènes dans ces nouveaux formats : la représentation des concepts de liste d’affichage, de dictionnaire ainsi que la représentation des mises à jour.

6.3.1 MPEG-4 BIFS

Nous nous sommes tout d'abord intéressés au format MPEG-4 BIFS qui nous paraissait le plus proche du format Flash. Nos travaux ont été publiés sur ce sujet à la conférence CORESA [09] et dans la revue T-CSVT [03].

6.3.1.1 Problèmes et solutions

Nous proposons de représenter la scène initiale, c'est-à-dire la feuille de prise de vue initiale, en primitives BIFS selon la structure XMT-A décrite dans le Code 6.1.

```
<Group DEF="root">
  <children>
    <Switch whichChoice="-1">
      <choice>
        <Group DEF="Formes">
          <children>
            <Shape DEF="FormeVide"/>
          </children>
        </Group>
        …
      </choice>
    </Switch>
    <OrderedGroup>
      <children>
        <Transform2D DEF="Couche1">
          <children><Shape USE="FormeVide"/></children>
        </Transform2D>
        <Transform2D DEF="Couche2">
          <children><Shape USE="FormeVide"/></children>
        </Transform2D>
        …
      </children>
    </OrderedGroup>
  </children>
</Group>
```

Code 6.1 – Exemple de représentation d'une scène initiale de dessin animé dans le format XMT-A

Cette représentation utilise un arbre de scène à deux branches. La première, matérialisée par le nœud `Switch`, permet de définir une branche de l'arbre BIFS qui ne sera pas rendu à l'écran, ici grâce à l'attribut `whichChoice` dont la valeur est `-1`. Les éléments définis dans cette branche ne seront jamais visibles sauf s'ils sont référencés ailleurs dans l'arbre, en utilisant la primitive `USE` décrite dans le chapitre 3. C'est le cas par exemple de la forme géométrique nommée `FormeVide`. La seconde branche est matérialisée par un nœud `OrderedGroup` qui définit un sous arbre dont les enfants correspondent aux différentes couches de la prise de vue, ici matérialisées par les noeuds `Transform2D`. Chaque couche, identifiée par un nom de la forme "`Couche`*`i`*", est vide au départ. L'animation de la feuille de prise de vue est décrite par un ensemble de mise à jour BIFS. Les éléments sont ajoutés ou enlevés au dictionnaire, plus particulièrement à la branche "`Formes`", comme illustré dans le Code 6.2.

```
<par begin="0.0">
  <Insert atNode="Formes" atField="children" position="END">
    <Transform2D DEF="Forme1">
      …
    </Transform2D>
  </Insert>
</par>
<par begin="10.0">
   <Delete atNode="Forme1"/>
</par>
```

Code 6.2 – Représentation d'une mise à jour du dictionnaire dans le format MPEG-4 XMT-A

Un élément du dictionnaire est inséré dans la liste d'affichage par remplacement du ou des enfants d'une couche "`Couchei`" par une référence (`USE`) sur l'élément en question. On peut également modifier les propriétés d'affichage de chaque couche pour simuler les changements de caméra (*zoom*, *pan*) en modifiant les attributs de transformation du noeud "`Couchei`". Ces deux opérations sont décrites en XMT-A dans le Code 6.3.

```
<par begin="2.0">
  <Replace atNode="Couche1" atField="scale" value="1.5 1.5"/>
  <Replace atNode="Couche1" atField="translation" value="-302.0 960.0"/>
  <Replace atNode="Couche1" atField="children" position="0">
    <Transform2D USE="Forme1"/>
  </Replace>
</par>
```

Code 6.3 – Représentation de mises à jour de la liste d'affichage dans le format MPEG-4 XMT-A

Pour finir, l'agrégation de la scène initiale décrite dans le Code 6.1, des mises à jour de manipulation du dictionnaire décrites dans le Code 6.2 et de celles de la liste d'affichage décrites dans le Code 6.3, aboutit à un document XMT-A décrivant un flux BIFS représentant un dessin animé vectoriel.

6.3.1.2 Résultats et problèmes

Grâce à ces techniques de représentation, et après avoir standardisé dans la norme BIFS de nouveaux outils, notamment de nouvelles primitives permettant les transformations matricielles de couleur (nœud `ColorTransform`) et de déformations (nœud `TransformMatrix2D`), nous avons pu, avec succès, réaliser un convertisseur de dessins animés Flash au format MPEG-4 BIFS.

Nous avons pu exploiter les capacités intrinsèques de *streaming* des flux BIFS et pour la première fois, diffuser des dessins animés au format vectoriel, tout en maintenant une synchronisation avec un flux audio, en utilisant les outils traditionnels de *streaming* : le format de fichier MP4, les protocoles RTSP/RTP et le serveur de *streaming Darwin Streaming Server*.

A la suite de cette expérience, nous nous sommes rendu compte que cette représentation ne satisfaisait pas à nos critères, notamment en compression, en consommation mémoire et en complexité surtout

pour les terminaux à faibles capacités. Une des optimisations que nous avons donc introduites dans cette représentation par rapport au format Flash a été l'introduction de commandes BIFS pour détruire les objets devenus inutiles (`Delete`). En effet, le format Flash n'est pas un format prévu pour le *streaming*. De ce fait, il ne permet pas une gestion fine de la mémoire utilisée par la scène. Il est bien sûr possible en analysant le fichier complet, de déterminer la durée de vie d'un objet dans le dictionnaire et de libérer la mémoire nécessaire à cet objet au moment opportun. Cependant, dans un contexte de *streaming*, il est nécessaire d'indiquer explicitement l'instant où un objet n'est plus nécessaire, car on ne sait pas a priori si l'objet sera utilisé dans le restant du flux et il n'est pas possible de réaliser d'analyse complète a priori. Nous verrons également dans la suite de ce chapitre que cela a un impact important sur la consommation mémoire, ce qui peut être crucial notamment sur les terminaux à faible capacité.

6.3.2 SVG

Nous nous sommes également intéressés au format SVG pour représenter les scènes animées car nous souhaitions expérimenter l'utilisation des animations SMIL et évaluer le comportement des lecteurs de contenus SVG jouant des contenus fortement animés.

6.3.2.1 Représentation SVG de dessins animés vectoriels

Nous nous sommes tout d'abord intéressés à la manière dont nous pouvions représenter les dessins animés en utilisant le modèle de document XML sans nous soucier des problématiques de *streaming*. Nous avons pris comme hypothèse de travail que le lecteur chargerait tout le document XML avant de lire le contenu (avec tous les inconvénients que cela comporte). Ces travaux ont été présentés à la conférence SMIL Europe 2003 [07].

Nous avons utilisé la même représentation arborescente à deux branches que précédemment. La notion de dictionnaire est une notion reconnue dans la norme SVG et est matérialisée par l'élément `defs`. Il permet de stocker des objets graphiques sans qu'ils soient visibles. Dans cette représentation SVG, nous avons donc défini tous les objets du dessin animé dans une branche dont la racine est un élément `defs`. Pour l'autre branche, nous avons utilisé un élément groupant `g` dont les enfants, similairement à la solution BIFS, sont des primitives `use`. Un exemple de scène initiale est décrit dans le Code 6.4.

```
<svg width="384" height="288" viewBox="0 0 384 288"
   xmlns="http://www.w3.org/2000/svg"
   xmlns:xlink="http://www.w3.org/1999/xlink">
<defs>
   <g id="Forme1">…</g>
   …
  <g id="FormeN">…</g>
</defs>
<g>
  <use id="Couche1"/>
  <use id="Couche2"/>
```

```
  <use id="Couche3"/>
  <use id="Couche4"/>
  <use id="Couche5"/>
</g>
```

Code 6.4 – Représentation d'une scène initiale de dessin animé en SVG

Nous nous intéressons ici à la problématique de représentation de l'animation sans notion de mise à jour. Nous étudierons dans la section 6.3.3 l'utilisation de mise à jour LASeR pour représenter les animations. Sans notion de mise à jour, comme nous l'avons indiqué dans le chapitre 2, la seule possibilité déclarative et standard pour effectuer ces animations est l'utilisation de primitives `set` de la norme SMIL.

Nous avons donc défini une représentation SVG des mises à jour de scène à l'aide des primitives `set`, comme le Code 6.5 le décrit[8]. Ces mises à jour sont ensuite concaténées dans le fichier XML à la suite de la scène initiale avec les informations temporelles indiquant leur instant d'exécution. Nous les avons regroupées dans un élément `g` pour faciliter la compréhension, comme le montre l'exemple.

```
<g id="trame_1">
  <set begin="0.08" xlink:href="#Couche1" attributeName="transform"
   to="matrix(1.5,0.0,0.0,1.5,-302.0,960.0)"/>
  <set begin="0.08" xlink:href="#Couche1" attributeName="xlink:href"
   to="#Forme1"/>
  <set begin="0.08" xlink:href="#Couche2" attributeName="transform"
   to="matrix(1.1249847,0.0,0.0,1.1249847,-224.0,864.0)"/>
  <set begin="0.08" xlink:href="#Couche2" attributeName="xlink:href"
   to="#Forme2"/>
  <set begin="0.08" xlink:href="#Couche3" attributeName="transform"
   to="matrix(1.1249847,0.0,0.0,1.1249847,74.0,102.0)"/>
  <set begin="0.08" xlink:href="#Couche3" attributeName="xlink:href"
   to="#Forme3"/>
</g>
…
<g id="trame_M">…</g>
</svg>
```

Code 6.5 – Représentation de mises à jour de la liste d'affichage dans le format SVG

6.3.2.2 Résultats et problèmes

Bien que cette méthode permette la représentation des dessins animés au format SVG, nous avons noté une limitation importante : l'utilisation mémoire. En effet, l'obligation de définir tous les objets à l'instant initial oblige à avoir tous les objets en mémoire dès le début de la scène. De plus, à la fin du

[8] Cette utilisation de la primitive set n'est pas conforme actuellement à la norme SVG. Une autre représentation conforme mais plus compliquée est possible. Celle-ci est utilisée ici pour des raisons de simplification.

dessin animé, lors de la lecture de la dernière trame, le lecteur possède en mémoire tous les symboles du dictionnaire. L'occupation mémoire est donc très importante.

A la suite de ce constat, nous avons proposé et standardisé, dans la norme SVG, l'élément `discard` afin de résoudre ce problème. Cet élément est l'équivalent de la commande BIFS de destruction d'objets, mais formulé comme un élément d'animation SMIL dont l'instant d'activation correspond à l'instant d'exécution de la destruction de l'objet cible.

La représentation finale à laquelle nous avons abouti nous a permis de valider notre convertisseur de contenu Flash en contenu SVG. De plus, l'utilisation de l'élément `discard` nous a permis de maîtriser la consommation mémoire. Cependant, du fait de l'utilisation du langage XML, outre le problème de *streaming* pour lequel nous proposons une solution dans la dernière partie de ce chapitre, un problème de compression se pose, sachant que la compression ZLIB recommandée par la norme SVG n'est pas efficace pour une solution de *streaming*, comme nous l'avons indiqué dans le chapitre 3. Enfin, nous le verrons dans le chapitre suivant, un problème lié à cette représentation est également l'efficacité de la représentation choisie dans la phase de composition et rendu de la scène.

6.3.3 LASeR

La dernière expérience, que nous avons effectuée, a consisté à réaliser l'export de dessins animés vectoriels du format Flash au format LASeR. Comme nous l'avons indiqué dans le chapitre 1, LASeR se base sur le langage SVG pour les aspects graphiques. La représentation LASeR de la liste d'affichage et celle du dictionnaire sont donc strictement identiques à celles utilisées dans la représentation SVG.

Cependant, du fait que le langage LASeR utilise un réel mécanisme de mise à jour, nous avons utilisé cette fonctionnalité pour modifier la liste d'affichage (transformation et identifiants d'objets dans une couche) et nous n'avons pas utilisé d'animation SMIL. A l'inverse d'une animation SMIL qui modifie une surcouche de présentation de l'arbre DOM, la mise à jour LASeR modifie directement l'arbre DOM. De ce fait, une mise à jour LASeR ne requiert plus aucun traitement après son application, à l'inverse d'une animation SMIL qui nécessite au moins l'opération permettant de déterminer si elle est active. De ce point de vue, le langage REX pourrait également être utilisé en remplacement des mises à jour LASeR.

Enfin, après la phase de compression, le résultat final est donc un flux LASeR binaire et non un document SVG "zippé" qui allie les avantages des deux solutions précédentes (BIFS et LASeR).

6.3.4 Synthèse sur le choix de la structure d'une scène

Nos expérimentations, sur les dessins animés, nous ont permis de vérifier qu'il était possible de représenter les scènes multimédia animées complètes, produites à l'issue du processus de création

décrit précédemment, sous la forme d'un flux, dont la scène initiale est représentée par un arbre à deux branches comme nous l'avons indiqué et ce dans n'importe quel langage muni d'outils d'animation ou de mise à jour comme les langages BIFS, SVG et LASeR. Cependant, cette représentation est problématique par rapport aux critères que nous nous sommes fixés : un problème de compression et un problème de distribution dans un contexte de *streaming* et d'adaptation.

6.4 Codage efficace de scènes animées

Comme nous l'avons indiqué dans le chapitre 2, le débit utilisé par la description d'une scène peut être important en fonction de la taille de la scène. Dans le cas des dessins animés que nous avons décrits précédemment, il est effectivement important de mettre en place des solutions de compression efficace car les contenus graphiques vectoriels peuvent être très volumineux s'ils ne sont pas compressés (de l'ordre du mégabit par seconde). Nous avons vu dans le chapitre 3 que l'efficacité des mécanismes de compression de scènes se situait au niveau de la structure de la scène et au niveau des données et en particulier au niveau des objets graphiques. Dans la suite de cette partie nous présentons les expérimentations que nous avons effectuées dans ces domaines pour en déduire une synthèse sur les mécanismes à utiliser pour obtenir une compression efficace.

6.4.1 Codage de structure

6.4.1.1 Réduction de la redondance des objets graphiques dans le format BIFS

Nous avons expérimenté l'utilisation des techniques de codage de la norme BIFS, décrite dans le chapitre 3, pour le codage de contenus animés, dont la représentation est décrite précédemment. Nous avons rapporté une partie de ces travaux à la conférence CORESA [09] et dans la revue T-CSVT [03].

Les formes graphiques vectorielles dans les contenus auxquels nous nous sommes intéressés sont principalement des polygones ou des polylignes, qui se représentent dans le langage XMT-A respectivement par des nœuds `IndexedFaceSet2D` ou `IndexedLineSet2D`, comme le décrit le Code 6.6.

```
<Shape>
<appearance>
<Appearance DEF="ap1">
<material>
<Material2D emissiveColor="0.79607844 0.7882353 0.72156864" filled="true" >
<lineProps>
<LineProperties lineColor="0.0 0.0 0.0" width="0.0"/>
</lineProps>
</Material2D>
</material>
</Appearance>
</appearance>
<geometry>
<IndexedFaceSet2D colorPerVertex="false">
<coord>
```

```
<Coordinate2D point="1147.5 0.0 1147.5 … -838.0 0.0 0.0 1147.5 0.0"/>
</coord>
</IndexedFaceSet2D>
</geometry>
</Shape>
```

Code 6.6 – Exemple de représentation d'un polygone en XMT-A

Une analyse des statistiques (Tableau 6.1) de nos séquences de test montre la forte redondance dans la structure de la scène : le nombre de nœuds `Shape`, `Appearance` et `Material2D` est approximativement le même; et le nombre de nœuds `Shape` est approximativement égal à la somme du nombre de primitives `IndexedLineSet2D` et `IndexedFaceSet2D`.

Type de nœud / Séquence	Shape	Appearance	Material2D	LineProperties	IndexedFaceSet2D	IndexedLineSet2D	Coordinate2D	Tous
1	6095	6094	6094	5915	4421	1672	6093	37341
1a	6914	6861	6861	6073	2925	3713	6886	41782
2	5318	5317	5317	4809	2675	2641	5316	31824
2a	8695	8653	8653	8082	421	8164	8681	52613
3	3126	3125	3125	2937	148	2976	3124	18969
3a	6716	6715	6715	6195	3274	3440	6714	40409
4	7793	7792	7792	7381	5764	2027	7791	47041
4a	10310	10268	10268	9561	6809	3391	10296	62191
5	7072	7071	7071	6788	5486	1584	7070	42589
5a	5243	5242	5242	5098	3852	1389	5241	31996
6	1292	1291	1291	1287	156	1134	1290	8081
6a	4208	4207	4207	3957	2803	1403	4206	25520
7	2187	2186	2186	2098	1785	400	2185	13413
7a	3391	3390	3390	3354	2830	559	3389	20709
8	4006	4005	4005	3987	3368	636	4004	24587
8a	2694	2693	2693	2391	21	2671	2692	16297
9	3340	3339	3339	3128	104	3234	3338	20378
9a	38276	38265	38265	33992	15602	22663	38271	228296
10	681	680	680	602	416	263	679	4093
11	6592	6591	6591	4878	2212	4378	6590	37917
12	4644	4643	4643	3535	30	4612	4642	26800
13	13962	13961	13961	11887	5426	8534	13960	81973
14	16319	16318	16318	14075	3718	12599	16317	95986
15	4375	4374	4374	4358	3780	593	4373	26374
Total	173249	173081	173081	156368	78026	94676	173148	1037179
Moyenne	7219	7212	7212	6515	3251	3945	7215	43216

Tableau 6.1 – Nombres et types des nœuds BIFS utilisés dans les séquences de dessins animés

Une première méthode immédiate pour réduire la taille de cette représentation XMT est d'utiliser l'attribut MPEG-4 `use`, comme nous l'avons décrit dans le chapitre 3. Cet attribut permet de réutiliser des objets précédemment définis. L'utilisation de `use` dans les séquences de dessins animés est présentée dans l'exemple Code 6.7, le principe étant de créer une palette de nœuds `Appearance`. Dans cet exemple, l'apparence du polygone est spécifiée comme réutilisant une apparence définie précédemment dans la palette et nommée "`ap1`".

```
<Shape>
<appearance><Appearance USE="ap1"/></appearance>
<geometry>
<IndexedFaceSet2D colorPerVertex="false">
<coord>
<Coordinate2D point="-103.0 -290.0 -101.0 -284.0 …" />
</coord>
</IndexedLineSet2D>
</geometry>
</Shape>
```

Code 6.7 – Exemple d'utilisation de la primitive BIFS USE dans le cas d'un dessin animé

Bien que cette technique réduise la quantité d'informations de structure (et de données) à coder, on peut s'apercevoir que la représentation résultante en XMT-A est toujours volumineuse pour au final signaler un polygone, une référence vers une apparence et une liste de points. De plus, du fait de la bijection entre XMT-A et le format binaire BIFS, la même remarque peut être faite sur le format binaire.

Nous avons donc utilisé une autre technique en faisant appel à la notion de *Proto* présentée également dans le chapitre 3. Deux prototypes ont été créés pour décrire de manière plus compacte les polygones et polylignes. L'interface de ces prototypes expose uniquement l'apparence du contour, sous la forme d'un nœud `Appearance` (la notion de palette est toujours utilisable), et la liste des points sous la forme d'un vecteur de points. De ce fait, on supprime l'utilisation et le codage des nœuds `IndexedLineSet2D` ou `IndexedFaceSet2D` et `Coordinate2D`.

Nous expliquons ici le gain de codage obtenu. Les instances des prototypes remplacent un sous arbre de la scène qui nécessiterait 30 bits (données non comprises). La déclaration de chaque *Proto* coûte, elle, 81 bits. En effet, il faut coder l'interface du prototype en plus de son corps. Ensuite, l'utilisation de prototype implique de pouvoir les identifier dans la scène. Le coût de codage d'un prototype inclut également le nombre de bits nécessaire pour identifier les instances de *Proto* (*protoIDbits*). Le codage d'une instance de *Proto* coûte 14 bits auxquels s'ajoute l'identifiant du *Proto*, soit *protoIDbits*. L'utilisation des *Protos* s'avère gagnante quand la formule ci-dessous est validée, *N* étant le nombre de répétitions de la structure que remplace le *Proto*.

Dans les contenus auxquels nous nous intéressons, nous ne définissons que deux *Protos* (polygone, polyligne), *protoIDbits* est donc égal à 1. La représentation choisie permet donc de gagner en

compression si le nombre de structures redondantes N est supérieur à un nombre minimal N_{min} égal à 6 et le gain est alors de 15 bits par répétition. Or, cette condition est largement satisfaite d'après les statistiques indiquées dans le Tableau 6.1.

Le gain est ainsi important puisque le coût de codage de la structure d'une instance redondante est divisé par deux. Sur l'ensemble de séquences sur lequel nous avons travaillé, il est évalué à 10 % en moyenne par séquence. Il est intéressant de noter que la technique utilisée est déclinable également avec les technologies, comme XBL, présentées au chapitre 3. Cependant, compte tenu des statistiques de nos séquences de test (densité d'information supérieure à 80%), le gain obtenu, même s'il est intéressant, n'est pas suffisant pour réduire significativement la taille totale des fichiers. Nous avons donc également étudié l'utilisation de techniques de codage des données. Ces travaux sont présentés dans la section 6.4.2.

6.4.1.2 Codage statistique de scènes

A la suite des travaux sur la compression de la structure présentés dans la section précédente, nous nous sommes également intéressés à l'efficacité de la consommation mémoire des mécanismes de codage de structure. Les travaux de normalisation autour du format MPEG-4 LASeR ont montré qu'un des inconvénients du codage BIFS était la taille de décodeur BIFS. Sur les terminaux à très faible capacité mémoire, cela peut être un inconvénient pour le déploiement d'application.

La taille des décodeurs BIFS est due à l'utilisation du codage contextuel. Comme nous l'avons décrit dans le chapitre 3, un décodeur doit stocker des tables d'association (type du nœud parent, type du nœud enfant, mot de code binaire). De même, pour chaque champ codé, des tables indiquent les associations (nœud, champ, mot de code binaire) dans différents contextes de codage (arbre de scène initial, mise à jour, route).

Nous avons donc étudié la possibilité d'utiliser un codage nécessitant moins de tables de codage. Pour cela, nous avons utilisé une approche statistique et un codage de Huffman. Nous avons effectué une étude statistique sur les séquences de conformité de la norme LASeR. Cette analyse produit, pour toutes les séquences, les statistiques d'occurrence des éléments et des attributs LASeR/SVG sous la forme suivante :

- la fréquence d'occurrence d'un élément;
- la fréquence d'occurrence d'un attribut par élément;
- et la fréquence d'occurrence d'un attribut.

La différence entre la seconde et la troisième donnée est pertinente car un même attribut peut être spécifié sur plusieurs éléments différents. A la suite de cette analyse, que nous avons contribuée au groupe MPEG, il apparaît clairement que certains éléments (`polyline`, `polygon`, `g`, `path` …) et

certains attributs (`stroke`, `points`, `fill`, `id` …) sont très fréquents. A partir de ces résultats, nous avons donc généré les codes de Huffman correspondant aux fréquences d'occurrences obtenues. Nous avons ensuite proposé une syntaxe binaire simple permettant de coder des scènes SVG basées sur ces codes de Huffman. Cette syntaxe est décrite dans la Figure 6.3. Nous avons ensuite réalisé un encodeur/décodeur suivant cette syntaxe.

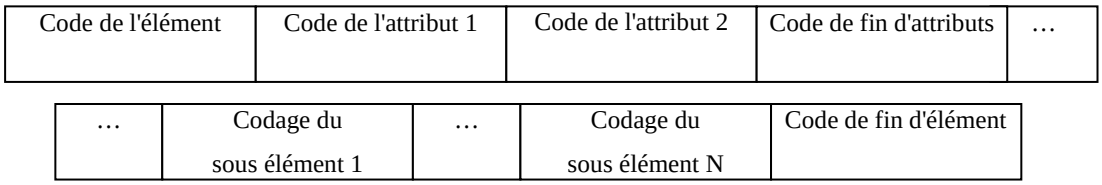


Figure 6.3 – Codage d'un élément XML utilisant des codes de Huffman

Après expérimentation, nous avons rejeté l'utilisation de codes de Huffman différents pour un même attribut pour des éléments différents pour deux raisons : d'une part, le stockage des tables par élément serait aussi coûteux en mémoire que le codage BIFS, et d'autre part, les gains obtenus n'étaient pas significatifs. Ceci s'explique par le fait que la majorité des attributs en SVG est spécifiable sur tous les éléments du langage.

Notre proposition consiste donc à utiliser un codage de Huffman avec deux tables : une table de codes pour les éléments et une table de codes pour les attributs. Le Tableau 6.2 indique, pour les séquences de conformité LASeR, le gain en compression obtenu avec la syntaxe décrite dans la Figure 6.3 par rapport au codage LASeR.

Gain	Structure uniquement	Structure et données
Moyenne	59,1%	23,1%
Moyenne pondérée	75,8%	16,2 %

Tableau 6.2 – Comparaison de l'encodage des séquences LASeR

Notre syntaxe apporte un gain de 75,8 % en moyenne, moyenne pondérée par la taille du flux compressé sans données. Ce gain s'élève à 16,2% en moyenne pondérée par la taille du flux compressé données comprises.

Ces résultats montrent que le codage statistique que nous proposons améliore significativement l'efficacité du codage de structure dans les documents LASeR, avec une limitation au niveau de l'extensibilité. En effet, il faudra un nombre de bits, pour coder un nouvel élément de la nouvelle version du langage, supérieur ou égal au nombre de bits nécessaire pour coder l'élément le moins fréquent de la version précédente du langage.

6.4.1.3 Synthèse sur la proposition du codage des structures de scènes

Notre proposition pour le codage efficace de la structure des descriptions de scènes, dont la représentation est décrite dans la section 6.3, se décompose donc en trois points :

- suppression du codage contextuel et utilisation d'une unique table de codage pour coder les éléments et les attributs afin de réduire la taille du décodeur;
- utilisation d'un codage statistique de type Huffman pour augmenter l'efficacité du codage. On peut éventuellement, si le langage est évolutif, réserver des mots de code pour les éléments définis dans de futures versions du langage;
- et utilisation de prototype de codage pour supprimer les structures redondantes de la scène. Cela modifie le codage précédent en ajoutant un ou plusieurs symboles fortement probables. Pour cela, nous proposons de réserver un ou plusieurs mots de code de Huffman permettant de coder ces prototypes. Nous proposons également la réservation des mots de codes pour le codage des attributs des prototypes.

6.4.2 Codage des données

La partie précédente s'est intéressée à l'amélioration du codage des structures des descriptions de scènes. Dans cette partie, nous présentons une analyse de l'efficacité des mécanismes de codage existants sur les données et plus précisément sur celles des contenus vectoriels animés, du type dessins animés. Nous présentons ensuite nos propositions d'améliorations en la matière.

Dans les scènes animées que nous avons traitées, mais cela s'applique également à d'autres contenus volumineux comme les contenus cartographiques, la majeure partie des données provient des objets graphiques vectoriels et ces données sont principalement des données de types :

- triplet de valeurs décimales représentant des couleurs, souvent dans l'espace colorimétrique RGB ou sRGB. Les valeurs sont souvent normalisées entre 0 et 1.
- valeur scalaire décimale représentant les épaisseurs de traits,
- et liste de paires de coordonnées décimales correspondant aux sommets des polygones et polylignes composant les dessins.

Le Tableau 6.3 indique les statistiques de ces données dans les séquences de dessins animés que nous avons étudiées.

Pour ces trois types de données, comme nous l'avons décrit dans le chapitre 3, la norme MPEG-4 propose des outils de codage, avec ou sans perte. Cependant, certains de ces outils ne sont optimaux que dans certaines situations. Nous présentons ici le résultat d'une étude que nous avons réalisée afin de déterminer la méthode de codage la plus appropriée.

Séquence	Nombre de triplets de valeurs décimales	Nombre de valeurs scalaires	Nombre de paire de coordonnées
1	6 522	5 921	219 192
1a	4 359	3 387	291 644
2	4 239	4 809	143 619
2a	9 646	2 507	363 222
3	2 617	2 937	135 739
3a	5 001	4 225	241 087
4	6 767	7 381	206 705
4a	13 526	2 106	258 286
5	6 328	6 788	191 800
5a	4 428	5 072	202 960
6	2 351	1 287	51 387
6a	3 133	2 097	196 915
7	3 843	2 098	80 835
7a	3 054	2 097	113 815
8	3 727	4 038	72 287
8a	1 866	703	113 688
9	2 751	3 128	139 392
9a	38 041	35 978	671 772
10	1 046	602	24 323
11	7 034	4 878	84 249
12	1 803	3 535	111 094
13	9 499	11 887	170 228
14	10 449	14 075	152 032
15	8 119	4 358	78 066
Total	160 149	135 894	4 314 337
Moyenne	6 673	5 662	179 764

Tableau 6.3 – Statistiques des données numériques dans les séquences de dessins animés

6.4.2.1 Utilisation du codage sans perte

Nous avons tout d'abord analysé l'efficacité de l'outil de codage sans perte "*Efficient Float Coding*" dans le cas des contenus de types dessin animés. Dans ces contenus, comme le décrit le Tableau 6.3, les points occupent la majeure partie des données. Ces points sont positionnés sur une grille de résolution de 2^{-N} pixel, et leurs coordonnées sont comprises 0 et $2^{M}-1$. On les notera sous la forme suivante : $k_1 + k_2.2^{-N}$. Le Tableau 6.4 indique la taille maximale occupée par une coordonnée décimale, avec le mécanisme "*Efficient Float Coding*".

M \ N	**5**	**4**	**3**	**2**	**1**	**0**
10	25	24	22	21	19	17
8	23	22	20	19	17	15
6	21	20	18	17	15	13
4	19	18	14	15	13	11
2	17	16	12	13	11	9

Tableau 6.4 – Nombre maximal de bits pour coder un nombre rationnel selon le procédé « Efficient Float Coding »

Par exemple, pour le codage des nombres décimaux compris entre 0 et 1023, dont la partie décimale est un multiple de 1/16, le tableau indique qu'on utilisera au maximum 24 bits. Par ailleurs, d'après l'Équation 3.1, on sait qu'il faudra au minimum 4 bits (notamment pour coder la valeur 0). Par les calculs, on montre que le nombre moyen de bits pour coder ces nombres est de 2 bits inférieur au nombre maximal de bits du Tableau 6.4. En prenant en compte le fait que le codage d'un nœud `QuantizationParameter`, pour signaler l'utilisation de cette technique pour tout l'arbre, ne coûte que 16 bits, il résulte que l'utilisation de ce codage permet de réduire le nombre de bits nécessaires pour coder les nombres décimaux dans la scène.

Cette technique est donc intéressante pour réduire sans perte le nombre de bits nécessaires pour coder les points décrivant les objets vectoriels. Cependant, nous verrons dans la partie suivante, que les mécanismes de quantification permettent un codage plus efficace avec une dégradation en qualité maîtrisée.

6.4.2.2 Utilisation de mécanismes de quantification

Comme nous l'avons vu dans le chapitre 3, les langages existants BIFS, Flash et LASeR permettent l'utilisation de mécanismes de quantification pour coder certains types de données et dans certaines parties de l'arbre de scène. Cependant, comme nous l'avons décrit, ces mécanismes doivent être utilisés avec précaution pour le codage des scènes vectorielles car celles-ci étant susceptibles d'être visualisées avec différents zooms, des artefacts de codage peuvent apparaître si les paramètres de quantification sont mal choisis. La quantification d'un contenu graphique vectoriel suppose donc de déterminer la résolution à partir de laquelle on considère les dégradations acceptables.

Pour cela, il est intéressant de visualiser les artefacts de codage, décrits dans le chapitre 3, sur des contenus réels, afin de voir l'impact sur la qualité visuelle du contenu. Nous présentons sur un exemple les résultats obtenus par l'utilisation des mécanismes de quantification BIFS afin de souligner les problèmes rencontrés dans la recherche du meilleur compromis compression/qualité. Nous proposons ensuite des méthodes pour résoudre ces problèmes.

6.4.2.2.1 Exemple de quantification d'un objet graphique vectoriel

La Figure 6.4 illustre un symbole issu d'un dessin animé vectoriel. Les coordonnées des points de ce dessin sont exprimées dans le repère dont l'origine est le coin supérieur gauche de l'image, dont l'axe des abscisses est orienté vers la droite et l'axe des ordonnées est orienté vers le bas. Elles sont comprises en abscisse entre 0.0 et 658.0 et en ordonnée entre 0.0 et 471.2. La précision des points est inférieure ou égale au $20^{ème}$ de pixel. Il faudra donc théoriquement en moyenne 15 bits pour représenter le dessin sans dégradation visible.

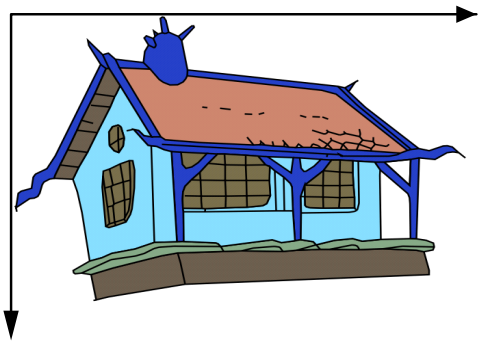


Figure 6.4 – Exemple d'objet graphique vectoriel issu d'un contenu vectoriel animé

Le Tableau 6.5 donne le résultat du codage de ce symbole suivant la norme BIFS en fonction du nombre de bits *N* utilisé pour coder les coordonnées des points du dessin. La première colonne donne l'image résultante après quantification et échantillonnage au pixel près. La seconde colonne représente la différence entre les images produites avec et sans quantification. La troisième colonne indique le PSNR résultant de cette différence. L'utilisation de cet outil présente l'avantage de pouvoir comparer nos résultats avec des codages d'image naturelle. Les deux dernières colonnes du tableau indiquent la taille du contenu codé avec quantification et le facteur de compression par rapport au contenu codé, toujours en BIFS, mais sans quantification en utilisant 32 bits par coordonnée. Ces chiffres sont à comparer avec la taille totale du contenu codé sans quantification qui est de 9407 octets, dont 6680 octets sont utilisés pour coder 835 points, qui représentent 71% du contenu.

Il faut noter également que nous avons effectué le même travail sur une centaine d'objets graphiques issus de contenu Flash et que tous les résultats (compression ou PSNR) sont similaires à ceux présentés ci-dessous.

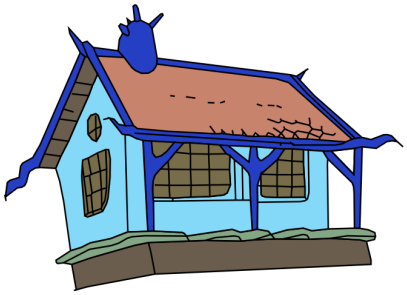
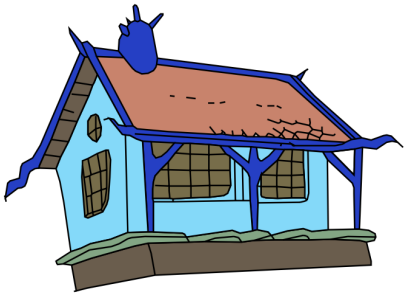
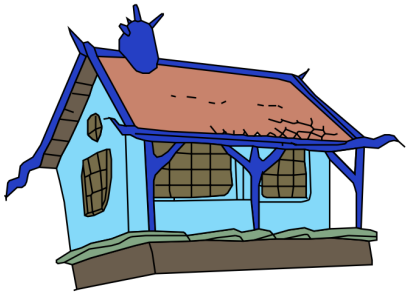
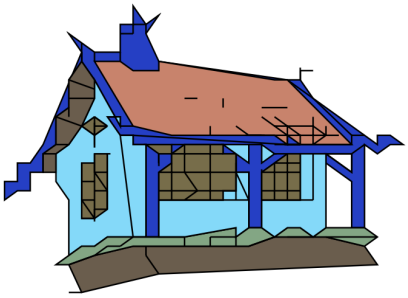
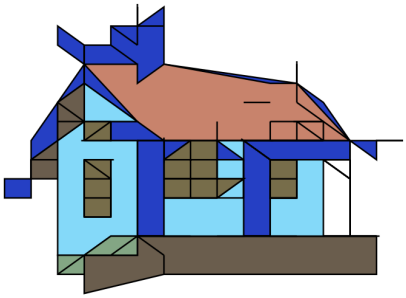
N	Image résultante	Différence	PSNR	Taille	Compression
14			55,87	3450	63,33%
9			28,52	2406	74,42%
8			23,29	2198	76,63%
5			13,33	1571	83,30%
4			11,86	1363	85,51%

Tableau 6.5 – Compression et qualité d'un objet graphique vectoriel selon la norme BIFS

Ce tableau indique qu'il y a peu de différence entre le contenu codé avec une valeur théorique de *N* égale à 14 bits et les contenus codés avec *N* compris entre 10 et 13. Le PSNR reste au dessus de 30 dB. Cela est dû à deux facteurs : tout d'abord, tous les points ne nécessitent pas le nombre de bits théorique de 15 pour être représentés sans perte, et ensuite, pour ces valeurs de *N*, la résolution varie entre 0,04 et 0,64 pixel. Or, l'échantillonnage des images pour le calcul du PSNR étant réalisé au pixel, le PSNR n'est que peu affecté.

Pour *N* égal à 9 bits, la résolution est égale à 1,28 et donc les dégradations commencent à être importantes mais l'anticrénelage limite encore les effets de la dégradation (sans anticrénelage, le PSNR pour la même valeur de *N* vaut 26.64 dB). Cette valeur de *N* semble donc la meilleure valeur pour coder ce dessin. A titre d'information, le nombre de bits nécessaires pour coder cette image selon le format JPEG, avec un facteur de qualité de 80%, est de 36 Ko, 15 fois supérieur à celui utilisant le codage vectoriel quantifié.

Par contre, pour des valeurs de *N* inférieures à 9, la qualité visuelle du dessin est altérée significativement par la quantification. Le PSNR chute en dessous de 30 dB. Une des conséquences de la quantification grossière est l'apparition de trou entre les différents contours. Cela est du au fait qu'un même point appartenant à deux contours différents peut être quantifié de manières différentes selon les paramètres choisis pour chaque contour.

6.4.2.2.1.1 Commentaires sur le PNSR

Il faut être prudent avec les chiffres de PSNR indiqués ci-dessus car ils correspondent à un échantillonnage de l'image avec une résolution au pixel près. Si on *zoom* sur l'image (en avant ou en arrière), puis si on régénère des images à partir des informations vectorielles et enfin si on calcule à nouveau ces PSNR, les résultats diffèrent. En effet, on remarque (Figure 6.5) que le PSNR gagne 2 à 3 dB quand la résolution de l'image double, à niveau de quantification constant, et ce, de manière constante avec le niveau de quantification, sauf quand la quantification produit une scène trop dégradée (*N* inférieur ou égal à 7). Cela s'explique par le fait que les erreurs de quantification affectent les contours, en les déplaçant, or plus l'image est grande moins les contours sont déplacés, donc plus l'erreur de codage est faible. Un autre facteur affectant le PSNR est la différence de couleur entre deux zones voisines. Si un contour est déplacé et si les couleurs, de part et d'autre du contour, sont proches, le PSNR ne sera pas beaucoup affecté.

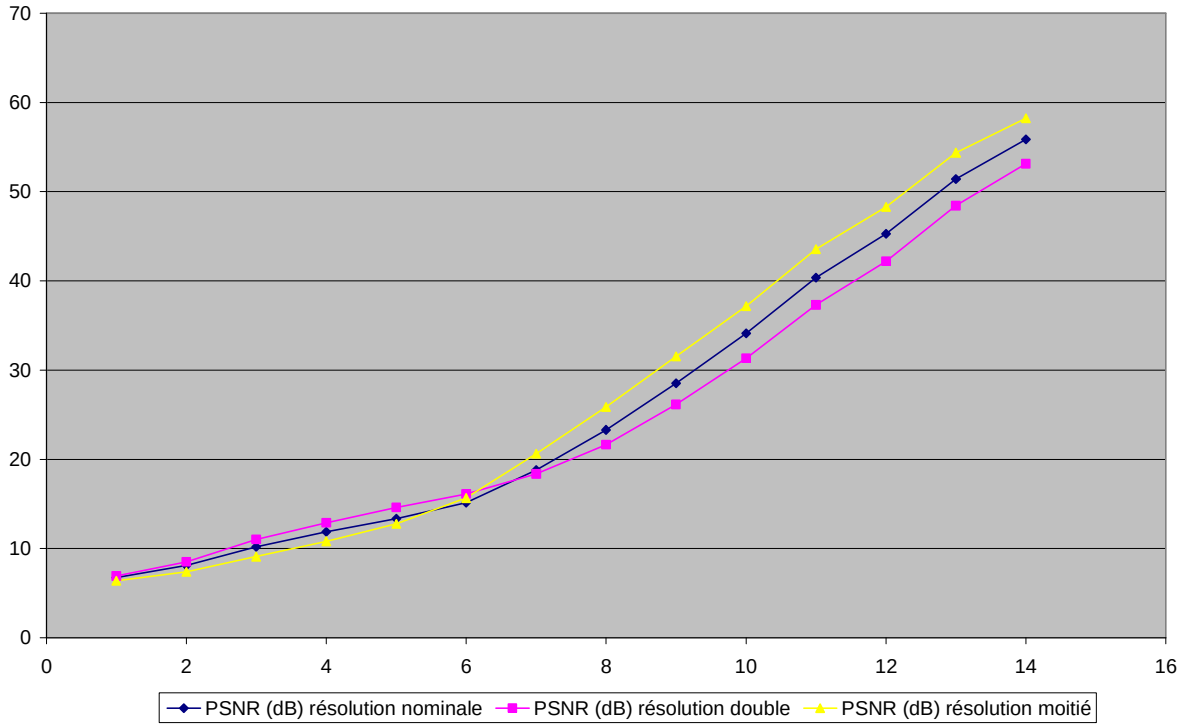


Figure 6.5 – PSNR (dB) de la Figure 6.4 en fonction paramètre de quantification et de la résolution

6.4.2.2.2 Propositions d'améliorations

Nous proposons ici une méthode qui consiste à modifier l'organisation des données pour éviter les artefacts de quantification. En VRML, tout comme en BIFS, il est possible de représenter un contour par une liste d'entiers correspondant aux index des points dans une liste de points préalablement définie. Pour cela, on définit un nœud `Coordinate2D` qui est réutilisé (via la primitive `USE`) par plusieurs nœuds `IndexedLineSet2D` et `IndexedFaceSet2D`. Le champ `coordIndex` de ces nœuds est utilisé pour spécifier les index dans la liste de points du nœud `Coordinate2D`.

Dans les objets graphiques qui nous intéressent, qui sont formés de nombreux contours (fermés ou ouverts), c'est-à-dire de nombreux nœuds `IndexedLineSet2D` et `IndexedFaceSet2D`, la plupart des points sont réutilisés dans des portions de contours que partagent deux formes voisines, comme nous l'avons décrit dans le chapitre 1. Or pour être réutilisables, les points de deux contours voisins doivent être dans une même liste, dans un nœud `Coordinate2D`. De proche en proche, on se rend compte que pratiquement tous les points d'un objet comme celui de la Figure 6.4 doivent être contenus dans une même liste, dans un même nœud `Coordinate2D`, pour être réutilisables dans tous les contours de l'objet, c'est-à-dire dans tous les nœuds `IndexedLineSet2D` et `IndexedFaceSet2D`.

Cette représentation présente un intérêt de codage quand, en moyenne, le codage d'un point et de ses réutilisations, sous forme d'index sur cette liste globale de points, est plus efficace que le codage répété du point, ce qui est notre cas d'après les statistiques du Tableau 3.1.

Cette représentation présente également un second avantage. En effet, si tous les points d'un même objet graphique appartiennent à une même liste, on ne peut que les quantifier selon les mêmes paramètres. De ce fait, un point qui appartient à deux contours différents ne sera quantifié qu'une seule fois. On évite ainsi l'apparition de trous entre les contours.

6.4.2.2.3 Granularité de la signalisation

La quantification BIFS nécessite un nœud `QuantizationParameter` qui est codé dans le flux binaire en utilisant entre 100 et 200 bits. Il est donc impossible de placer un tel nœud devant chaque primitive graphique, et inefficace d'en placer un seul pour toutes les primitives graphiques qui forment l'objet composite. Il faut trouver le bon compromis. Nous avons montré par le calcul dans la revue T-CSVT [03] qu'il était possible de manière incrémentale de trouver le gain apporté par un nouveau nœud de quantification afin de déterminer lors du codage s'il est utile ou non.

6.4.3 Synthèse et proposition de codage

Notre proposition pour le codage des données des descriptions de scènes de type dessin vectoriel 2D, quel que soit le format utilisé, consiste donc :

- à utiliser une représentation des objets vectoriels basée sur l'utilisation d'une liste de points et d'index sur cette liste;
- à coder les points avec un mécanisme de quantification à mi-chemin entre la quantification BIFS et LASeR. Nous proposons l'utilisation de la formule de quantification BIFS en contraignant à l'utilisation d'une différence $v_{max} - v_{min}$ du type 2^k, afin de permettre le décodage sur les plateformes matérielles utilisant une représentation des nombres décimaux à virgule fixe.
- à ne pas utiliser de techniques de codage prédictif ou arithmétique, tels qu'ils sont décrits dans la norme BIFS, pour trois raisons : la complexité trop importante du décodage, l'inefficacité sur les contenus vectoriels (car le modèle d'approximation linéaire est trop simple) et la sensibilité aux erreurs.
- et enfin, à utiliser un mécanisme de signalisation de la quantification selon le modèle BIFS, c'est-à-dire permettant une quantification différenciée pour chaque objet.

6.5 Distribution efficace de scènes animées

Dans les parties précédentes de ce chapitre, nous avons décrit des méthodes pour représenter les scènes animées et compresser ces scènes de manière efficace. Cependant, nous avions laissé momentanément de côté le problème de la distribution de ces scènes lorsqu'elles ne sont pas compressées, c'est-à-dire lorsqu'elles sont exprimées en XML, comme c'est le cas pour les scènes SVG. Nous présentons dans cette partie, les études que nous avons réalisées pour améliorer la distribution de contenu SVG. Nous nous intéressons tout d'abord au téléchargement progressif puis au *streaming*. Nous proposons enfin une classification des modes de distribution de scènes animées s'appuyant sur une scène SVG.

6.5.1 Distribution progressive de documents XML

Comme nous l'avons indiqué dans le chapitre 5, la norme SVG permet le rendu progressif de scènes SVG. Cette technique permet notamment au lecteur SVG de présenter la scène reçue, par exemple via HTTP, avant qu'elle ne soit reçue entièrement, réduisant ainsi la latence de présentation. Afin de permettre un téléchargement progressif efficace, il faut qu'une première scène soit présentable rapidement après le début du téléchargement. Cette contrainte modifie la représentation SVG de la scène que nous avons présentée dans la section 6.3.2. En effet, celle-ci se base sur l'utilisation d'un unique dictionnaire dans lequel tous les éléments utilisés par la liste d'affichage sont présents dès le début de la scène. Cela implique que tous les objets, notamment vectoriels, soient définis avant même la description de la liste d'affichage initiale. Ceci n'est pas satisfaisant. Nous avons donc modifié cette représentation en profitant de la possibilité offerte par SVG pour définir un dictionnaire par trame. Ainsi, nous définissons un dictionnaire minimal initial et un dictionnaire par trame qui ne contient que les objets utilisés dans cette trame. Un exemple d'une telle représentation est décrit dans l'exemple Code 6.8.

```
<g id="trame_k">
  <defs>
    <!--définition des objets utilisés dans cette trame et les suivantes-->
    <g id="FormeK">…</g>
  </defs>
  <set begin="k" xlink:href="#CoucheL" attributeName="transform" to="…"/>
  <set begin="k" xlink:href="#CoucheL" attributeName="xlink:href"
   to="#FormeK"/>
</g>
```

Code 6.8 – Représentation améliorée des mises à jour au format SVG pour le téléchargement progressif

Cette représentation modifiée améliore le comportement dans un scénario de téléchargement progressif car le lecteur peut commencer à rendre la scène avant d'avoir tous les objets en mémoire. De plus, dans le cas du chargement progressif, la consommation mémoire n'est pas maximale au début de la lecture.

Bien que cette représentation couplée avec une distribution par téléchargement progressif soit plus satisfaisante, elle n'est toujours pas suffisante : le serveur ne peut pas contrôler le débit de ce qu'il envoie car il ne connaît pas l'association entre les données envoyées et l'instant auquel elles doivent être présentées à l'utilisateur. C'est typiquement ce que les solutions de *streaming* permettent de faire. Nous nous sommes donc intéressés à trouver une méthode pour diffuser en mode *streaming* des documents SVG. Nous présentons cette méthode dans la suite de ce chapitre.

6.5.2 Fragmentation temporelle de contenu SVG

Un document SVG est composé de déclarations d'objets audio-visuels, de déclarations d'animation, de déclarations d'écouteurs et de gestionnaires d'évènements. Dans les documents animés, toutes ces déclarations sont liées à la notion de temps de scène. Les animations ne sont actives qu'à partir d'un certain instant, comme nous l'avons décrit dans le chapitre 1. De même, les objets audio-visuels ne sont visibles qu'à partir d'un certain instant. Enfin, les écouteurs, les gestionnaires d'évènement ne sont utiles également qu'à partir d'un instant précis. Il est possible comme le montre la Figure 6.6, d'associer un temps de composition à chaque élément SVG.

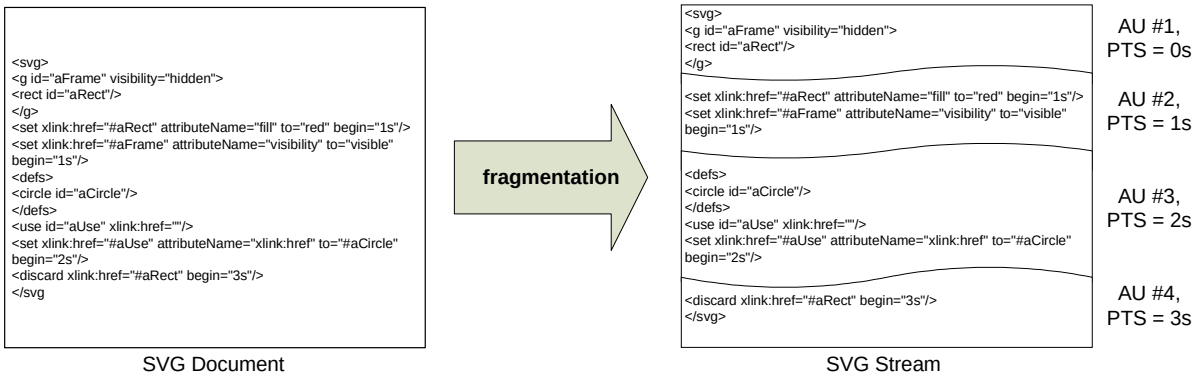


Figure 6.6 – Fragmentation temporelle d'un document SVG

Si on associe à chaque élément son instant de composition respectif, le document SVG se décompose en blocs d'éléments ayant le même instant de composition. Nous définissons qu'un document SVG est fragmentable temporellement :

- si le document est décomposé en plusieurs blocs distincts dont tous les éléments ont le même instant de composition;
- et si ces blocs sont, en suivant l'ordre du document, ordonnés de façon croissante selon leur instant de composition,
- ou s'il est possible de permuter ces blocs pour obtenir que les blocs soient présents de façon croissante dans le document, sans changer le résultat visuel de sa présentation.

Avec cette définition, un document fragmentable temporellement est un document à partir duquel on peut former un flux. En effet, chaque fragment temporel d'un document fragmentable temporellement correspond à une unité d'accès du flux.

6.5.3 Outil générique de fragmentation et de *streaming* XML

Afin de valider la possibilité de diffuser en *streaming* des contenus SVG fragmentables, nous avons ajouté dans l'outil `MP4Box`, un multiplexeur pour fichier MP4, la possibilité de fragmenter un fichier XML et d'importer les fragments comme unité d'accès. Pour cela, nous proposons un langage simple, en XML décrivant la fragmentation du fichier XML source. Le Code 6.9 donne un exemple de fichier XML dans ce langage.

```
<Stream timeScale="12.0" DTS_increment="1" baseMediaFile="flash2.svg">
<Sample isRAP="yes" xmlFrom="doc.start" xmlTo="frame_1.start"/>
<Sample isRAP="no" xmlFrom="frame_1.start" xmlTo="frame_1.end"/>
<Sample isRAP="no" xmlFrom="frame_2.start" xmlTo="frame_2.end"/>
<Sample isRAP="no" xmlFrom="frame_3.start" xmlTo="frame_3.end"/>
<Sample isRAP="no" xmlFrom="frame_4.start" xmlTo="frame_4.end"/>
<Sample isRAP="no" xmlFrom="frame_5.start" xmlTo="frame_5.end"/>
<Sample isRAP="no" xmlFrom="frame_6.start" xmlTo="frame_6.end"/>
<Sample isRAP="no" xmlFrom="frame_7.start" xmlTo="frame_7.end"/>
<Sample isRAP="no" xmlFrom="frame_8.start" xmlTo="frame_8.end"/>
<Sample isRAP="no" xmlFrom="frame_9.start" xmlTo="frame_9.end"/>
<Sample isRAP="no" xmlFrom="frame_10.start" xmlTo="frame_10.end"/>
<Sample isRAP="no" xmlFrom="frame_10.end" xmlTo="doc.end"/>
</Stream>
```

Code 6.9 – Exemple de fichier permettant la fragmentation temporelle d'un fichier XML

Ce code indique au logiciel `MP4Box` de créer un flux à partir du fichier "flash2.svg" dont les unités d'accès seront séparées d'1/12^{e} de secondes. Les unités d'accès sont repérées par l'élément `Sample`. Pour chaque unité d'accès à créer dans le flux, on indique : si elle doit être considérée comme un point d'accès aléatoire (ici seulement la première); le début et la fin de chaque fragment de document XML. Nous avons utilisé une syntaxe basée sur les évènements SAX pour indiquer ces débuts et fins. Ainsi, dans cet exemple, le premier fragment temporel commence au début du document et se termine quand l'élément dont l'identifiant "`frame1`" commence.

Grâce à cette technique, nous avons pu importer les fragments du document dans un fichier multimédia du type MP4, qui a ensuite été préparé pour le *streaming* (création des paquets RTP à partir des unités d'accès d'un flux en fonction du type de flux et de la taille maximale des paquets autorisés sur le réseau). Nous avons utilisé l'encapsulation en paquets génériques MPEG-4, spécifiée dans le document [77]. Enfin, nous avons utilisé le serveur de *streaming* de fichier MP4, *Darwin Streaming Server* [103], pour servir notre fichier selon le triplet de protocoles RTSP/SDP/RTP, que nous avons pu jouer avec succès dans le lecteur SVG que nous avons développé et qui est décrit dans le chapitre suivant. Enfin, nous avons pu également tester l'entrelacement de flux audio, vidéo et SVG

dans un même fichier MP4 que nous avons préparé pour le téléchargement progressif selon la structure décrite précédemment.

6.5.4 Efficacité du *streaming* de document SVG

Nous avons souhaité mesurer l'efficacité de cette méthode de *streaming* de contenu SVG par rapport aux méthodes actuelles (lecture locale et téléchargement progressif), en prenant en compte deux paramètres : le temps de chargement avant l'affichage de la première scène et l'utilisation mémoire. Nous avons utilisé plusieurs contenus vectoriels animés que nous avons joués selon les scénarios suivants :

- Scénario 1 : le contenu est lu à partir d'un fichier local, entièrement; et seulement après, la scène est visualisée.
- Scénario 2 : le contenu est lu à partir d'un fichier local et visualisé progressivement.
- Scénario 3 : le contenu est lu à partir d'un fichier distant en utilisant le protocole HTTP (sur un réseau avec un débit supérieur au débit moyen) et progressivement visualisé.
- Scénario 4 : le contenu est préparé suivant notre méthode et servi par un serveur de *streaming*, sur le même réseau.

Les résultats des temps d'attentes initiaux, en millisecondes, sont donnés dans le Tableau 6.6. Ils ont été vérifiés également avec d'autres contenus du même type.

Scénario	Contenu court Durée 14s Débit moyen 530 kbps Débit maximum 1590 kbps	Contenu intermédiaire Durée 87 s Débit moyen 869 kbps Débit maximum 3487 kbps	Contenu long Durée 140 s Débit moyen 817 kbps Débit maximum 3486 kbps
Scénario 1	890	22120	32350
Scénario 2	240	390	380
Scénario 3	440	530	470
Scénario 4	330	290	350

Tableau 6.6 – Temps d'attente initial (ms) avant visualisation de la première scène

Les deux premières lignes du tableau confirment évidemment que le contenu chargé et visualisé progressivement offre une amélioration par rapport à la lecture complète. De plus, elles montrent que l'impact de la taille du fichier sur le temps de chargement est important, voire inacceptable pour des fichiers dont la durée est moyenne : il faut 22 secondes de chargement du fichier pour visualiser un contenu de 87 secondes. La troisième ligne du tableau confirme également que selon le débit disponible pour la lecture du contenu (ici le réseau à un débit maximum inférieur au débit en lecture locale à partir du disque dur), le temps d'attente de la première visualisation est plus long. Enfin, il est

important de remarquer que le scénario de *streaming* et celui de téléchargement progressif aboutissent à des temps d'attente équivalents. Cela dit, ces temps dépendent des paramètres du lecteur de contenu (temps de mise en *buffer* pour le *streaming*, temps de chargement avant la première lecture). Afin de départager les deux scénarios, il est intéressant d'étudier l'occupation mémoire du lecteur dans les différents scénarios.

Nous avons mesuré la mémoire utilisée pour jouer le contenu durant la réception et la lecture. La Figure 6.7 montre l'utilisation de la mémoire en fonction du temps pour les différents scénarios précédents (pour le contenu court). Un nouveau scénario est introduit, il s'agit du scénario 0. Celui-ci est strictement équivalent au scénario 1 d'un point de vue distribution mais le contenu a été modifié pour supprimer l'utilisation des éléments SVG `discard`, afin d'illustrer la mémoire maximale nécessaire pour jouer le contenu. Les axes temporels des différentes figures sont alignés sur l'instant de visualisation de la première image, instant appelé t_0.

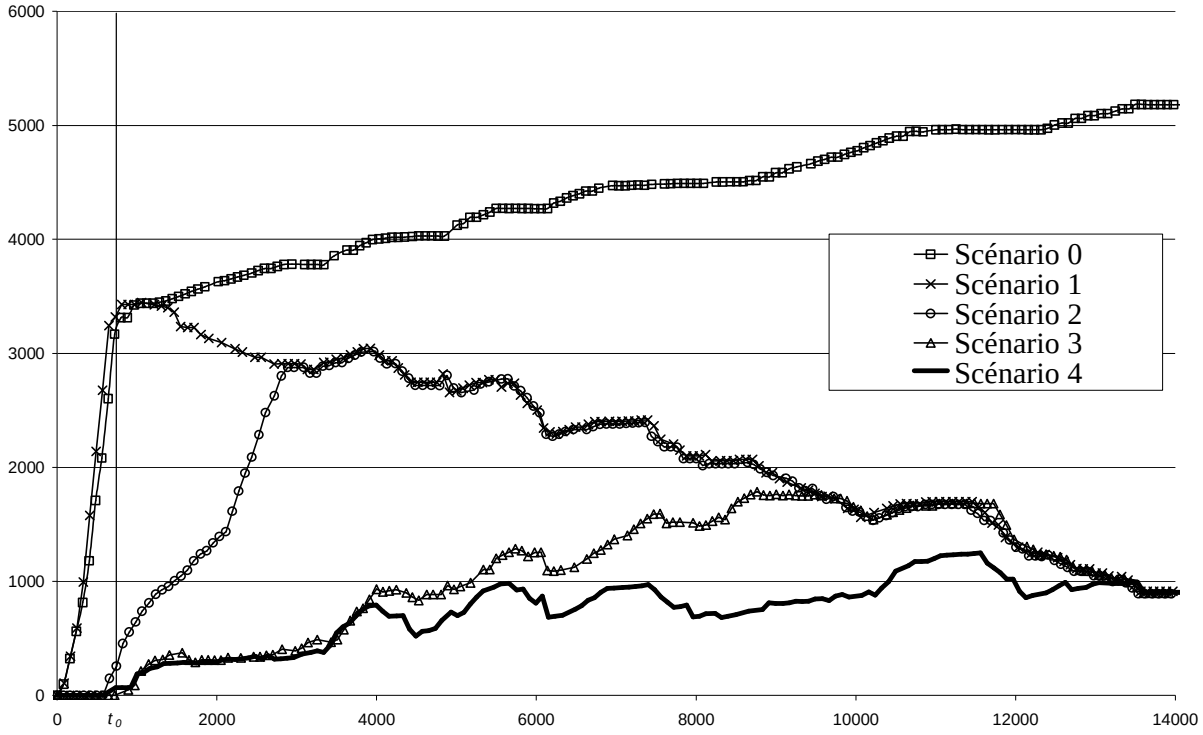


Figure 6.7 – Utilisation mémoire (en octets) pour la lecture d'un contenu SVG selon le mode de distribution en fonction du temps de scène (en millisecondes)

On remarque que, dans le scénario 0, la consommation mémoire augmente très rapidement jusqu'à t_0, de manière similaire au scénario 1. Il s'agit de la phase de lecture où toute la scène SVG est alors chargée en mémoire, y compris les parties qui ne seront jouées qu'à la fin de la séquence. La différenciation entre les scénarios 0 et 1 a lieu ensuite pendant la visualisation du contenu. Dans le scénario 0, l'affichage des images suivantes entraîne une consommation mémoire toujours croissante (constante à la fin). Cette croissance est due à l'allocation des structures nécessaires à la composition

qui ne sont jamais libérées car le lecteur ne sait jamais quand un objet est devenu inutile. En revanche, dans le scénario 1, le pic de consommation mémoire est atteint à t_0 car ensuite les structures inutiles sont libérées en fonction des éléments `discard`.

Les scénarios 2 et 3 conduisent à un comportement similaire. La consommation augmente moins rapidement que pour le scénario 0 du fait de la visualisation qui a lieu en parallèle, exécutant les éléments `discard`. On remarque que plus lente est la vitesse de lecture, moins le pic de consommation mémoire est élevé. Enfin, on remarque surtout que le pic minimum de consommation mémoire correspond au scénario 4 quand les structures nécessaires à la composition d'une image sont chargées juste avant la composition de l'image et libérées dès qu'elles sont devenues inutiles.

Ces résultats montrent l'avantage de l'élément `discard` qui réduit, à lui seul, l'occupation mémoire maximale nécessaire pour jouer le contenu en entier. Dans notre exemple, la mémoire maximale nécessaire est divisée, au minimum, par un facteur 1,5 en comparant les scénarios 0 et 1.

Le téléchargement progressif permet de visualiser le contenu avant que celui-ci ne soit complètement (télé)chargé. Cependant, un compromis doit être trouvé. Il est évident que si la vitesse de chargement est trop faible, le lecteur n'aura pas assez de données pour assurer une visualisation fluide. A l'inverse, comme le montre la Figure 6.7, si la vitesse de chargement est trop rapide, le lecteur consommera une mémoire plus importante que nécessaire. Le meilleur compromis consiste à contrôler temporellement le chargement des trames et à utiliser un format de stockage sous forme de flux (protocole de *streaming* ou fichier multimédia). Le téléchargement progressif allié à une diffusion contrôlée temporellement permet l'affichage fluide du contenu tout en réduisant la consommation mémoire, par un facteur 5, dans notre exemple.

Enfin, il est important de noter que, pour que la mémoire consommée à la fin de la lecture du contenu soit comparable à celle consommée quand le lecteur ne lit aucun contenu, il est nécessaire de supprimer tous les éléments de la scène SVG : c'est-à-dire les éléments graphiques, les animations déterminant les débuts et fins de chaque image, les éléments `defs` servant de dictionnaires, les éléments `g` délimitant les images. On voit ainsi une des limites de la représentation des séquences vectorielles animées en SVG. Il faut en effet explicitement supprimer les commandes de mise à jour de la scène. La solution LASeR, utilisant des commandes explicitement détruites après exécution, est de ce fait plus efficace.

6.5.5 Synthèse

Nous avons montré l'efficacité de notre proposition de *streaming* de contenu SVG. Cette proposition permet la lecture efficace de contenu SVG, en mode *streaming*, sans modification du lecteur SVG. Cette méthode reste valable pour le *streaming* de document XML générique mais à caractère temporel, avec une seule base de temps, et visualisable en mode progressif.

Cependant, du fait de la contrainte sur l'ordre du document et l'ordre des unités d'accès, il existe des contenus SVG qui ne se prêtent pas à la fragmentation temporelle et au *streaming*. Dans cette partie, nous complétons donc notre proposition avec une classification des méthodes de distribution des contenus multimédia basés sur une scène SVG. Notre classification s'articule selon trois axes :

- la taille du document SVG;
- l'utilisation de média audio-visuels;
- et la possibilité de fragmentation temporelle du document.

6.5.5.1.1 Classification des contenus SVG et méthodes de distribution

Le Tableau 6.7 décrit notre classification des méthodes de transmissions de contenu multimédia basé sur une description SVG.

Caractéristiques du document SVG			Méthode de distribution adéquate
Fragmentation temporelle possible	Utilisation de média	Taille du document	
Non	Non	-	Le document SVG ne peut pas être fragmenté en un flux donc seul le téléchargement du fichier est possible. Il n'est possible d'améliorer la distribution du document qu'en utilisant le téléchargement progressif.
Non	Oui	Faible	Le document étant de taille faible, le choix de la méthode de distribution de la scène n'a pas d'impact. Par contre, pour accélérer la mise en place de la scène, celle-ci peut être transmise en même temps que la configuration des flux média.
Non	Oui	Importante	Le document étant volumineux mais non fragmentable, il n'est distribuable que sous forme de fichier. Les médias, eux, peuvent utiliser n'importe quelle méthode (fichier, flux). La solution la plus appropriée est l'utilisation de méthodes distinctes pour le document (téléchargement progressif) et les médias (*streaming*).
Oui	-	-	La méthode la plus appropriée (pour la consommation mémoire) est le *streaming* dans un cas de lecture

			distante, notamment pour les fichiers volumineux. Le *streaming* du document peut avoir lieu conjointement au *streaming* des médias, dans un même multiplexe, en fonction des contraintes de synchronisation de la scène et des média. Dans le cas d'une lecture en local, la lecture d'un fichier SVG stocké dans un fichier de type MP4 est également la méthode la plus appropriée car elle permet une maîtrise de la consommation mémoire.

Tableau 6.7 – Classification des contenus SVG et mode de distribution

6.6 Représentation adaptable de flux de scènes animées

Le début de ce chapitre a présenté les bases d'une structuration des descriptions de scènes animées. Il a présenté également des méthodes pour compresser et distribuer de manière efficace ces contenus. Dans cette dernière partie, nous proposons une amélioration de ces outils pour prendre en compte des contraintes de scalabilité.

Au cours des projets ISIS [82][06] et DANAE [81][04], nous nous sommes intéressés également à la capacité d'adaptation d'un flux de scènes, selon la norme MPEG-21 [41]. Nous avons présenté, dans le chapitre 6, un bref rappel sur les notions d'adaptation et de scalabilité selon cette norme. Dans ce chapitre, nous décrivons nos travaux et réalisations. Ces travaux ont été publiés lors de la conférence MCube 2004 [04] et aboutissent à la proposition d'une organisation des flux de scènes basés sur des mises à jour afin de donner au flux une propriété de scalabilité.

6.6.1 Méthodes de scalabilité pour un flux de scènes animées

Adapter un flux scalable par rapport à des contraintes signifie supprimer des éléments du flux pour satisfaire ces contraintes. Les contraintes que nous avons tenté de satisfaire sont les contraintes de débit et de complexité de la scène. Nous souhaitons pouvoir transmettre des contenus vectoriels animés sur des réseaux à faible bande passante ou pour des terminaux à faible capacité de calcul. Nous avons donc travaillé sur les dessins animés tels que nous les avons décrits précédemment, pour lesquels nous avons développé deux techniques d'encodage scalable.

6.6.1.1 Adaptation par réduction du débit

Etant donnée la répartition de débit entre les informations de structure et les données dans les scènes et en particulier les dessins animés, l'adaptation par réduction du débit n'est pertinente que si on adapte les données. Les données auxquelles nous nous sommes intéressées sont a priori peu redondantes, la

réduction des données implique une dégradation de la qualité visuelle. L'enjeu est de trouver un bon compromis entre réduction de débit et perte de qualité.

Nous avons choisi de réduire le débit des dessins animés en réduisant le nombre de points utilisés pour représenter les primitives graphiques. Nous proposons une organisation des mises à jour de la scène en couches logiques. Dans cette proposition, la couche de base du flux contient les primitives graphiques avec un nombre minimal de points. Les couches d'améliorations contiennent des commandes d'insertion des points supplémentaires. Les figures Figure 6.8, Figure 6.9, Figure 6.10 et Figure 6.11 illustrent les couches de base et d'amélioration : le paramètre *ratio* représente le rapport entre la taille du fichier adapté et celle du fichier non scalable.

Figure 6.8 – Couche de base (ratio : 0.49)

Figure 6.9 – Couche de base et première couche d'amélioration (ratio : 0.57)

Figure 6.10 – Couche de base et deux couches d'amélioration (ratio : 0.92)

Figure 6.11 – Couche de bases et toutes les couches d'améliorations (ratio : 1.35)

Nous avons expérimenté deux techniques pour déterminer les points à placer dans les couches d'amélioration :

- une technique basée sur l'étude de la courbure au niveau d'un point, le principe étant que les points où la courbure était importante apportaient plus d'information que les autres;

- et une autre technique basée sur des calculs d'aires et de la contribution de chaque point à l'aire du polygone. Pour chaque point P_n du contour, nous calculons l'aire du triangle $P_{n-1}P_nP_{n+1}$. Nous conservons les points P_n pour lesquels l'aire du triangle $P_{n-1}P_nP_{n+1}$ contribue le plus à l'aire du polygone. Cette technique s'applique aux polygones convexes.

Pour évaluer l'efficacité des caractéristiques de scalabilité, il faut comparer différents aspects du codage. Dans le domaine de la vidéo, un codage scalable est acceptable s'il n'induit pas de ou s'il induit un faible surcoût de codage et s'il permet une granularité fine tout en fournissant une qualité équivalente au codage non scalable. Les travaux sur la réduction du nombre de points constituent une première étape mais comme le montre la Figure 6.12, cette technique de scalabilité introduit un surcoût de codage, que l'on peut évaluer à 35 % pour 7 couches d'améliorations, qui reste encore élevé.

De plus, la réduction du nombre de points permet de réduire le débit jusqu'à 35 % sans introduire d'artéfacts visuels trop importants mais pour une réduction de débit supérieure à 40 % les artefacts restent encore trop importants.

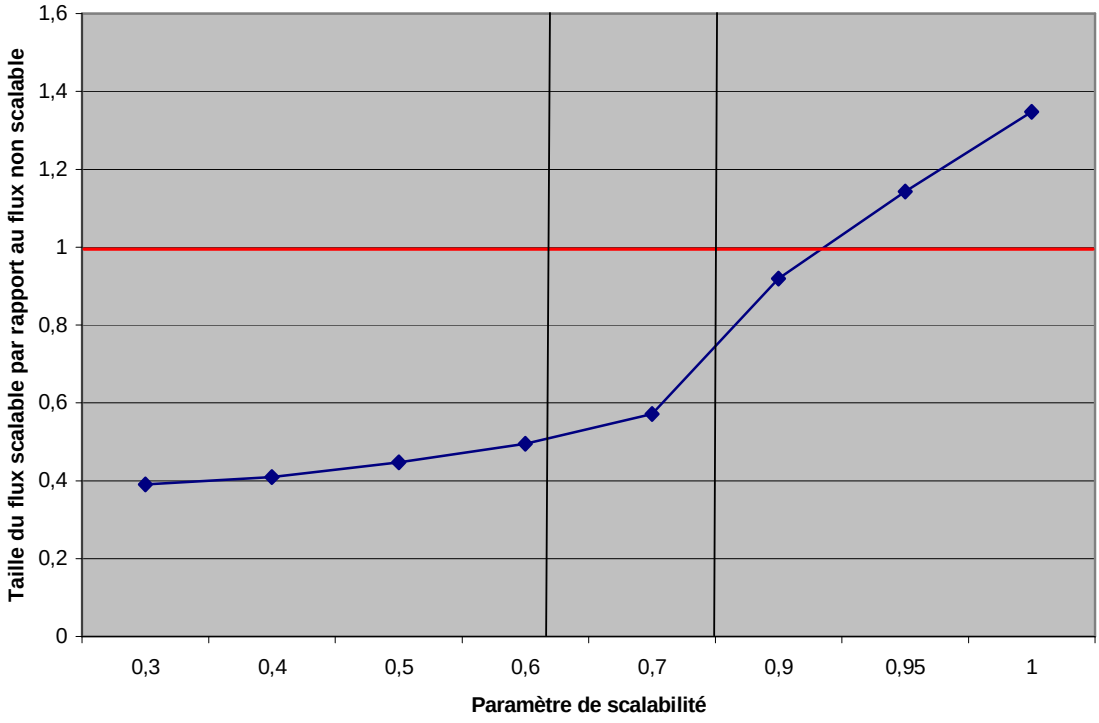


Figure 6.12 – Evaluation du surcoût de codage pour 7 couches d'améliorations

6.6.1.2 Adaptation par réduction de la complexité

Nous avons donc envisagé une seconde méthode. Les tests que nous avons pu effectuer sur des terminaux à faible puissance de calcul ont montré qu'il était souvent difficile pour un lecteur

multimédia de jouer des contenus vectoriels animés complexes. Le problème principal que rencontrent ces lecteurs est la complexité du rendu graphique des contours avec anticrénelage. A la différence d'un rendu bitmap, c'est-à-dire d'une image type JPEG, pour chaque contour d'objet, il faut calculer un polygone représentant le contour cet objet en prenant en compte l'épaisseur. Il faut ensuite effectuer une opération de fusion (*alpha blending*) du contour et du fond de l'objet pour afficher le contour sans effet de crénelage. Nous avons expérimenté des ralentissements de la vitesse de rendu allant d'un facteur 1 à 4 pour le rendu d'un même contenu, dans le même format, avec ou sans contour, sur un PDA de type IPAQ sans accélération graphique. A partir de cette expérience, nous avons donc mis en place une technique de codage scalable à 2 niveaux de scalabilité : avec ou sans contour. Un exemple de résultat est donné dans la Figure 6.13. Nous avons envisagé deux façons de procéder :

- En créant des commandes d'insertion de primitives du type `IndexedLineSet2D` en BIFS. Cette méthode nécessite de décomposer chaque objet possédant un contour en 2 objets : l'un sans contour, l'autre pour le contour uniquement. Cette technique n'est intéressante que pour les traits de contour non remplis. Nous l'avons écartée.
- En créant des commandes de remplacement des épaisseurs de traits. La couche de base est modifiée pour que toutes les épaisseurs de trait soient nulles, la couche d'amélioration positionne l'épaisseur à la bonne valeur.

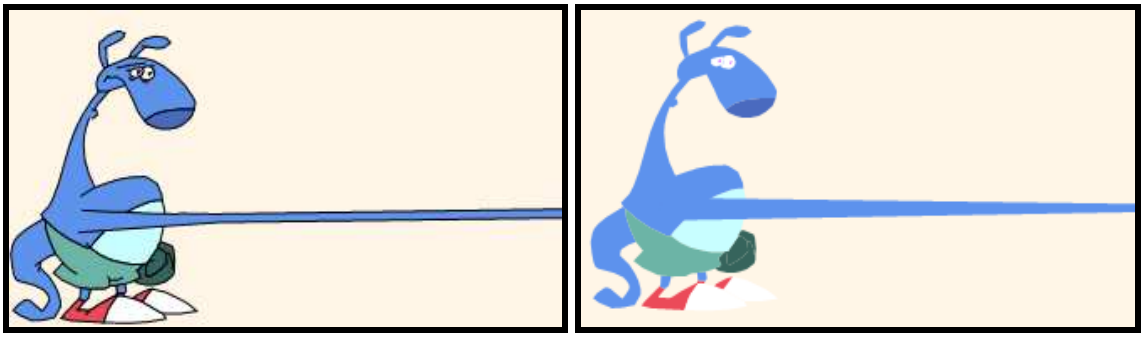


Figure 6.13 – Exemple de contenu vectoriel avec et sans tracé des traits de contours

Cette seconde méthode, contrairement à la précédente, permet de réduire à la fois la complexité au niveau du terminal. De plus, le surcoût, dans ce cas, est moindre car le remplacement d'une épaisseur de trait ne coûte qu'un faible pourcentage du débit comparé au codage des contours (liste de points). Cette technique est donc une approche de scalabilité de scènes vectorielles plus satisfaisante en terme de complexité et non en terme de débit. En revanche, du fait du caractère subjectif de l'évaluation de la qualité d'une telle couche de base, il est difficile de juger de manière définitive de l'acceptabilité d'un tel contenu par un utilisateur final.

6.6.2 Résultats

Suite à ces travaux, nous avons réalisé un encodeur BIFS capable de générer des flux BIFS suivant les méthodes de scalabilité décrites précédemment. Cet encodeur génère le flux BIFS mais également une description gBSD correspondante qui décrit la structure du flux BIFS en couches logiques. Un exemple de description gBSD est donné dans le Code 6.10.

```
<?xml version="1.0" encoding="UTF-8"?>
<dia:DIA xmlns="urn:mpeg:mpeg21:2003:01-DIA-gBSD-NS"
  xmlns:dia="urn:mpeg:mpeg21:2003:01-DIA-NS"
  xmlns:mpeg7="urn:mpeg:mpeg7:schema:2001"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<dia:Description xsi:type="gBSDType">
<Header>
<ClassificationAlias alias="M4B"
  href="urn:mpeg:mpeg4:bifs:cs:syntacticalLabels"/>
<DefaultValues addressUnit="byte" addressMode="Absolute"
  globalAddressInfo="samples\cartoons\bifs_aa001\aa001.bifs"/>
</Header>
<gBSDUnit start="0" length="4722" syntacticalLabel=":M4B:AccessUnit">
<Header>
<DefaultValues addressUnit="bit" addressMode="Absolute"/>
</Header>
<gBSDUnit start="0" length="997" syntacticalLabel=":M4B:Command"/>
<gBSDUnit start="997" length="36029" syntacticalLabel=":M4B:Command"/>
<gBSDUnit start="37026" length="33" syntacticalLabel=":M4B:Command"/>
<gBSDUnit start="37059" length="83" syntacticalLabel=":M4B:Command"/>
<gBSDUnit start="37142" length="83" syntacticalLabel=":M4B:Command"/>
<gBSDUnit start="37225" length="50" syntacticalLabel=":M4B:Command"
  marker="LOD-1"/>
<gBSDUnit start="37275" length="50" syntacticalLabel=":M4B:Command"
  marker="LOD-1"/>
<gBSDUnit start="37325" length="50" syntacticalLabel=":M4B:Command"
  marker="LOD-1"/>
<gBSDUnit start="37375" length="50" syntacticalLabel=":M4B:Command"
  marker="LOD-1"/>
<gBSDUnit start="37425" length="50" syntacticalLabel=":M4B:Command"
  marker="LOD-1"/>
…
```

Code 6.10 – Exemple de description gBSD décrivant un flux BIFS

Pour chaque unité d'accès BIFS, un élément `gBSDUnit` est créé avec pour label syntaxique `":M4B:AccessUnit"` permettant à un outil de fragmentation de document XML de découper ce fichier gBSD et de le stocker ou de le transmettre de manière synchronisée avec le flux BIFS correspondant. Pour chaque commande BIFS, un élément `gBSDUnit` est créé avec pour label syntaxique `":M4B:Command"`. De plus, un attribut `marker` indique si cette commande appartient à la couche d'amélioration.

Nous avons également réalisé une feuille de transformation XSLT pour réaliser l'opération de génération de la description gBSD du flux adapté. Cette feuille localise les éléments `gBSDUnit` qui

correspondent aux couches d'améliorations non souhaitées et calcule les longueurs des nouvelles AU après adaptation.

Enfin, l'encodeur que nous avons réalisé génère également un fichier XML contenant un descripteur MPEG-21 `AdaptationQoS` pour permettre au moteur d'adaptation de sélectionner l'index le plus élevé de la couche d'amélioration à conserver en fonction du débit disponible sur le réseau. Un exemple de descripteur `AdaptationQoS` est donné dans le Code 6.11. Il s'agit d'un document XML qui décrit la matrice d'association entre le vecteur débit (`BITRATE`) et le vecteur indiquant l'index de la couche d'amélioration la plus élevée (`LAYER`).

```
<DIA xmlns="urn:mpeg:mpeg21:2003:01-DIA-NS"
  xmlns:gbsd="urn:mpeg:mpeg21:2003:01-DIA-gBSD-NS"
  xmlns:mpeg7="urn:mpeg:mpeg7:schema:2001"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<Description xsi:type="AdaptationQoSType">
<Module xsi:type="LookUpTableType">
<Axis defaultIndex="0" iOPinRef="LAYERS">
<AxisValues xsi:type="IntegerMatrixType">
<Vector>0 1 2 3 4</Vector>
</AxisValues>
</Axis>
<Content defaultIndex="0" empty="false" iOPinRef="BITRATES">
<ContentValues xsi:type="FloatVectorType">
<Matrix mpeg7:dim="5 1">60580 102543 164821 212718 215154</Matrix>
</ContentValues>
</Content>
</Module>
<IOPin id="BITRATES" input="true" output="false"/>
<IOPin id="LAYERS" input="false" output="true"/>
</Description>
</DIA>
```

Code 6.11 – Exemple de description AQoS pour l'adaptation un flux BIFS

6.7 Synthèse

Nous avons présenté dans ce chapitre, un ensemble de propositions pour représenter, compresser et diffuser de manière efficace des descriptions de scènes représentant des contenus fortement animés, notamment du type dessins animés. Nous avons proposé une représentation de scènes arborescente mais structurée selon l'architecture dictionnaire, liste d'affichage. Nous avons également justifié l'utilisation de mises à jour de scènes pour représenter les animations, notamment pour la flexibilité de la manipulation de l'arbre de scène et pour la *streamabilité*. Nous avons présenté comment ces mises à jour pouvaient être utilisées pour permettre la scalabilité et la gestion du débit des descriptions de scènes. Enfin, dans le domaine de la compression, nous avons proposé une syntaxe pour coder la structure des descriptions de scènes de manière efficace. Nous avons également présenté comment encoder de manière optimale les objets graphiques vectoriels en utilisant une représentation précise et une quantification adaptée. Nous présentons dans le chapitre suivant, une proposition

d'implémentation d'un lecteur de scènes multimédia permettant de jouer ces contenus de manière efficace en termes de consommation mémoire et de puissance de calcul utilisée.

6.8 Résultats indirects

Durant cette thèse, nous avons été amenés à collaborer à l'amélioration du codage de document XML selon la norme MPEG-7 BiM. Grâce à l'analyse des mécanismes de quantification BIFS et notamment de la méthode et de la granularité de signalisation des paramètres de quantification BIFS, nous avons réalisé que le codage MPEG-7 BiM pouvait être amélioré dans le domaine du codage des données. L'amélioration apportée a consisté à définir des contextes d'encodage de l'arbre associé à un document XML et à associer à chaque contexte un encodeur particulier. Nous avons relié le contexte d'encodage au type de données. Ainsi, par exemple, tous les attributs de type chaîne de caractères sont définis comme appartenant à un contexte d'encodage particulier. On peut ensuite indiquer que l'on souhaite un encodage de type GZIP pour toutes ces chaînes de caractères. De même, on peut associer à toutes les valeurs décimales le contexte d'encodage utilisant une quantification linéaire. L'avantage de cette méthode est qu'elle permet à un créateur de langage XML de concevoir le schéma associé à ce langage de telle sorte que le codage des données soit optimal. Le résultat de ce travail a été valorisé dans un brevet [15].

Chapitre 7 Implémentation optimisée d'un lecteur de scènes multimédia animées et interactives

7.1 Introduction

Nous avons présenté, dans le chapitre précédent, un choix de représentation pour des scènes multimédia animées et interactive, ainsi que des outils de codage associés. Nous nous intéressons ici aux problèmes liés à la visualisation de ces scènes. En effet, parmi les terminaux de visualisation de scènes multimédia, les ordinateurs personnels représentent une large partie, mais les appareils mobiles (téléphones, consoles) et décodeurs de télévision numérique deviennent de plus en plus répandus et certains offrent la possibilité de jouer des contenus multimédia. Cependant, les terminaux de ces deux dernières catégories se différencient de la première catégorie car ils sont en général produits avec des contraintes fortes sur leur coût de revient et donc sur les capacités de calcul ou d'affichage et sur la mémoire disponible. Le problème de la présentation de contenu multimédia sur ces terminaux, étant données leurs contraintes matérielles, est d'offrir à l'utilisateur la meilleure qualité d'expérience possible. On pourra consulter la thèse de G. Di Cagno [32] pour avoir une définition précise de la qualité d'expérience d'un contenu multimédia.

Un des critères pour obtenir une bonne qualité d'expérience est la fluidité de la présentation du contenu multimédia. Lorsque la scène utilise des animations, notamment par interpolation, la fluidité est obtenue quand le nombre de cycles de présentation par seconde est important. Le nombre de cycles de composition nécessaire pour obtenir cette fluidité dépend de l'application envisagée. Par exemple, dans le cas de jeux vidéo, on recherche des fréquences de rafraîchissement supérieures à 60 Hz. Dans le cas de la présentation d'une scène multimédia, la fréquence de rafraîchissement nécessaire dépend du contenu. Certains contenus animés (bandeaux publicitaires) se satisfont d'une fréquence autour de 15 Hz. Une fréquence de rafraîchissement de l'affichage supérieure à 30 Hz est souvent retenue également, ce qui signifie que le temps de cycle complet (lecture, composition, rendu) doit prendre moins de 33 ms.

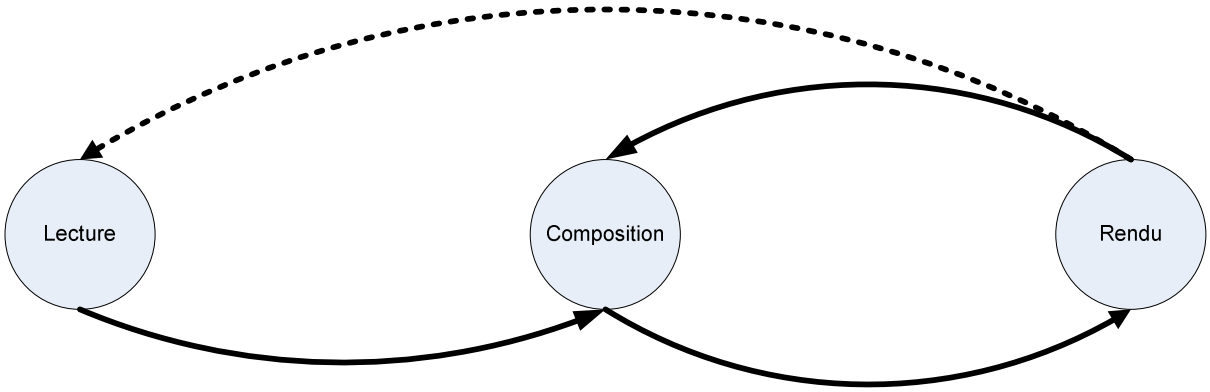


Figure 7.1 – Cycle de présentation d'une scène multimédia

Le cycle de présentation pour un contenu multimédia est illustré dans la Figure 7.1. Il se décompose en trois parties :

- la lecture de la totalité ou d'un fragment des données;
- la composition, à un instant donné, des différents éléments de la scène issus de la lecture;
- et le rendu de la scène composée, c'est-à-dire l'affichage des éléments visuels et le rendu sonore.

Le problème que nous tentons de résoudre dans ce chapitre est la conception d'un lecteur multimédia efficace pour différents types de terminaux, notamment pour des terminaux où la quantité de mémoire est limitée. Nous nous plaçons dans un contexte où la scène peut être modifiée par des mises à jour. Les approches d'optimisation qui consisteraient à analyser la scène pour en simplifier le traitement ne sont donc pas possibles car les hypothèses simplificatrices permettant des optimisations peuvent être contredites à n'importe quel instant par une mise à jour. Nous nous attachons donc à satisfaire deux objectifs : permettre une lecture fluide des contenus animés, en minimisant le temps d'exécution de chacun des processus du cycle de présentation; et limiter la consommation mémoire afin de pouvoir lire des contenus volumineux sur des terminaux contraints. Nous détaillons nos propositions au travers de la description de l'implémentation et l'intégration du support pour le langage SVG dans la plateforme logicielle GPAC [96].

7.2 Lecture optimisée de scènes SVG

Nous nous intéressons, dans cette partie, à analyser les contraintes principales de la phase de lecture des scènes multimédia concernant la rapidité d'exécution et la gestion mémoire. Ensuite, nous présentons comment améliorer ce processus. Nous proposons enfin une méthode de représentation mémoire, pour les objets SVG, permettant de minimiser l'occupation mémoire.

7.2.1 Rapidité de lecture

Le temps passé à la lecture de la description d'une scène est un facteur à prendre en compte à double titre. Si la lecture du contenu s'effectue par fragment, la phase de lecture fait alors partie intégrante du cycle de composition, il faut que le temps de lecture d'un fragment soit bien inférieur à la durée maximale du cycle (par exemple, 33 ms) pour laisser du temps aux phases de composition et de rendu. Si, en revanche, la lecture du contenu s'effectue en une seule étape, c'est-à-dire entièrement avant la première phase de composition, le temps de lecture ne participe pas au cycle de présentation, mais introduit un délai initial qu'il faut minimiser également.

Le problème de la rapidité de lecture est encore plus crucial quand la scène est exprimée en XML, comme c'est le cas pour de nombreux langages de descriptions de scènes (comme SVG). La lecture de document XML peut en effet être très lente et de nombreux travaux se sont penchés sur ce problème. On peut citer notamment les travaux autour du logiciel `XMLScreamer` [33] qui est un outil pour générer des lecteurs de document XML rapides. D'autres travaux s'appuient sur les techniques de compression présentées dans les chapitres précédents.

Nous avons souhaité évaluer cette deuxième approche, en particulier sur des scènes au format SVG. Pour cela, nous avons implémenté un lecteur de descriptions de scènes au format SVG capable de lire les données issues d'un fichier SVG (c'est-à-dire XML) et d'un flux binaire au format LASeR. Nous avons basé le lecteur de fichier SVG sur le logiciel `LibXML` [99]. Ce dernier est utilisé ici dans le but d'avoir une référence en la matière, même s'il ne s'agit pas du lecteur XML le plus rapide, comme les travaux sur `XMLScreamer` le soulignent.

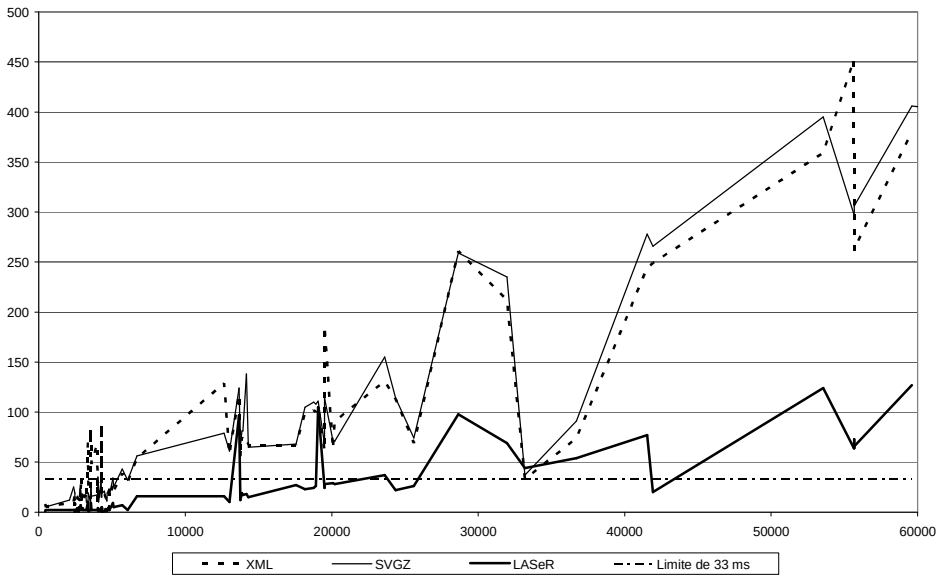


Figure 7.2 – Temps de chargement (ms) d'un contenu SVG en XML, XML compressé selon l'algorithme ZLIB et en LASeR en fonction de la taille du fichier XML (octets)

La Figure 7.2 montre le temps nécessaire à la lecture des descriptions de scènes SVG, issues des séquences de conformité de la norme LASeR, à partir d'un fichier XML, d'un fichier XML compressé selon l'algorithme ZLIB et d'un fichier binaire LASeR, en fonction de la taille du fichier XML. Les temps indiqués, en millisecondes, correspondent au temps de lecture du fichier complet. La lecture consiste à lire les données issues du fichier source et à les interpréter pour créer les objets dans l'arbre de scène. Dans les trois cas, les objets créés en mémoire sont identiques.

La Figure 7.2 illustre trois points :

- Le fait que la lecture de documents binaires est plus rapide que la lecture de documents textuels, et que, bien évidemment, cela s'applique aux descriptions de scènes. Elle montre, en particulier, que la lecture binaire au format LASeR est 5 fois plus rapide que la lecture au format SVG (compressé avec ZLIB ou non).
- Le fait que même dans le cas d'une lecture binaire, il existe néanmoins des contenus pour lesquels le temps de lecture est très proche voire supérieur aux 33 ms nécessaires pour un cycle complet de présentation à 30 Hz.
- Le fait que le temps de lecture/décodage d’une scène complète (cela s'applique également à un fragment de scène) peut-être très variable d’une scène à l’autre. Cela s'explique par le fait que les scènes multimédia utilisées décrivent un nombre d'objets, notamment graphiques, lui-même très variable. Ainsi, le temps de chargement d'une scène peut varier de quelques millisecondes à plusieurs secondes dans le cas de contenus volumineux (comme les contenus cartographiques).

Au vu de ces chiffres, dans une architecture de lecteur multimédia efficace, il est donc nécessaire pour la fluidité de lecture de recourir à un lecteur de descriptions de scènes binaires, surtout pour les descriptions de scènes volumineuses. De plus, nous avons vu dans le chapitre précédent qu'un décodeur de scènes binaires pouvait être réalisé avec une occupation mémoire faible. Cette consommation est notamment faible en comparaison avec la mémoire utilisée pour un lecteur XML, où les noms des éléments doivent être stockés de manière statique dans le lecteur. Ainsi, l'utilisation d'un lecteur de descriptions de scènes binaires est avantageuse à double titre : pour la consommation mémoire et la rapidité de lecture.

Cependant, certains contenus resteront trop volumineux (même compressés) pour être lus dans un temps raisonnable par rapport au cycle de rendu comme on peut le voir également sur la Figure 7.2. Pour qu'un lecteur multimédia puisse lire ces contenus de manière efficace, l'amélioration de la rapidité des algorithmes de lecture n'est pas suffisante. A notre sens, une architecture de diffusion de contenu multimédia, et notamment de contenu graphique vectoriel (cartographie, publicités animées) doit s'appuyer également sur deux points importants :

- L'intégration au niveau du lecteur de contenu d'un mode de visualisation progressive, comme nous l'avons décrit dans le chapitre précédent. Cette méthode n'est envisageable que pour des contenus adaptés, c'est-à-dire qui ont été construits pour permettre un affichage progressif cohérent.
- L'utilisation dès la phase de création de contenu, des mécanismes de fragmentation temporelle des contenus, plus précisément l'utilisation de la notion de flux de descriptions de scènes, pour découper la scène volumineuse en entités plus petites dont l'auteur pourra contrôler l'instant d'affichage de façon précise.

Suite à ces recommandations, nous écarterons donc les améliorations au niveau de la rapidité de lecture pour nous concentrer sur l'amélioration de la consommation mémoire.

7.2.2 Consommation mémoire

La phase de lecture de la description d'une scène a pour but de produire une représentation mémoire utilisable par le module de composition. Le choix d'une représentation mémoire pour les objets d'un langage de descriptions de scènes est primordial pour maîtriser la consommation mémoire. Trois points sont importants à souligner :

- Certaines scènes, comme celles de type cartographique par exemple, peuvent utiliser des centaines de milliers d'objets. Une représentation mémoire inefficace de ces objets conduira à une consommation mémoire excessive et inacceptable pour des terminaux contraints. Il faut donc prendre garde à optimiser la consommation mémoire de chaque type d'objet et notamment des types d'objets les plus fréquents. En effet, il est peu probable de rencontrer des scènes où des milliers d'objets vidéo seront utilisés, alors que des scènes avec des milliers d'objets graphiques vectoriels sont courantes.
- Le choix d'une représentation mémoire doit également tenir compte de la rapidité d'accès aux données notamment pour l'étape de composition. Par exemple, on pourrait décider de stocker la description de la scène en mémoire sous la forme binaire transmise, la consommation mémoire serait minimale mais chaque accès à un élément pendant la phase de composition nécessiterait un décodage et donc un temps d'accès très long. De plus, une modification de la scène par animation, mise-à-jour ou script nécessiterait un encodage.
- Enfin, le concepteur de lecteur multimédia dispose d'une certaine latitude dans le choix de la représentation mémoire des objets représentant la scène. Cette représentation peut être différente de celle utilisée pour créer la scène (représentation interne à l'outil auteur) ou pour la transmettre (forme binaire) mais elle ne doit pas être trop éloignée de la représentation de la scène définie par le langage de descriptions de scènes, afin que les interfaces programmatiques

d'accès à ces structures, comme DOM ou MicroDOM dans le cas de scènes SVG, soient toujours utilisables de manière efficace.

7.2.2.1 Première proposition : structuration suivant l'approche BIFS

Dans cette partie, nous décrivons une première approche qui a consisté à calquer la structuration des objets SVG selon celle utilisée pour le langage BIFS. Nous présentons également ses limites.

7.2.2.2 Description

Dans le langage BIFS, le tracé d'un rectangle "minimaliste", c'est-à-dire avec ses propriétés de remplissage et de contour utilisant les valeurs par défaut, est décrit dans le Code 7.1 .

```
<Shape><geometry><Rectangle size="100 100"/></geometry></Shape>
```

Code 7.1 – Rectangle BIFS minimaliste

Le nœud `Shape` indique qu'on souhaite afficher une forme à l'écran dont la géométrie est décrite par le nœud `Rectangle`. La taille du rectangle est donnée dans l'attribut `size`, et il s'agit de la seule propriété de l'élément `Rectangle`. Si on souhaite exprimer plus de caractéristiques (remplissage, contour, bord …), il faut utiliser plus de nœuds comme dans le Code 7.2.

```
<Transform2D translation="50 50">
 <children>
  <Shape>
   <appearance>
    <Appearance>
     <material>
      <Material2D emissiveColor="1 0 0" filled="true">
       <lineProps>
        <XLineProperties width="5" lineColor="0 0 1"/>
       </lineProps>
      </Material2D>
     </material>
    </Appearance>
   </appearance>
   <geometry><Rectangle size="100 100"/></geometry>
  </Shape>
 </children>
</Transform2D>
```

Code 7.2 – Rectangle BIFS détaillé

Cet exemple montre qu'on a besoin d'un nœud `Transform2D` pour indiquer les changements de position du rectangle, d'un couple de nœuds `Appearance` et `Material2D` pour indiquer la couleur de remplissage, d'un nœud `XLineProperties` pour indiquer l'épaisseur du contour et sa couleur.

Le langage BIFS s'appuie, comme on le voit dans les exemples, sur une structure de scène utilisant beaucoup de nœuds mais avec des nœuds simples, c'est-à-dire ne possédant pas beaucoup de propriétés. Cette structuration permet notamment de réutiliser des nœuds. Par exemple, si deux formes

géométriques ont les mêmes propriétés de contour, le nœud `XLineProperties` entier peut-être réutilisé en mémoire en utilisant le mécanisme `USE` présenté au chapitre 3. Une représentation en mémoire découle naturellement de cette structuration du langage. Cette représentation est décrite dans le Code 7.3.

```
typedef struct {
  SFVec2f propriété_1;
  SFFloat propriété_2;
  SFBool  propriété_3;
  …
} NoeudBIFS_1;
```

Code 7.3 – Représentation mémoire possible décrivant un nœud BIFS

Cette représentation mémoire possède deux avantages :

- Etant donné qu'un nœud ne possède que peu de propriétés, la structure est de taille réduite, et il est possible d'allouer ces propriétés (avec leur valeur par défaut) en même temps que l'allocation d'un nœud : l'allocation des structures de la scène est donc rapide car une seule allocation est effectuée.
- Comme les propriétés sont décrites dans le nœud sans utilisation de pointeur ni de liste, l'accès à ces propriétés est direct : il n'y a pas de test, ni de parcours de liste à effectuer.

L'inconvénient de cette approche concerne la consommation mémoire. En effet, celle-ci n'est pas optimale car les propriétés des nœuds sont systématiquement allouées même si elles ne sont pas présentes dans la scène lue. Cependant, dans le cas des scènes BIFS, cet inconvénient est négligeable car lorsqu'un nœud est utilisé, la plupart de ces propriétés sont utilisées.

Notre première approche a donc consisté à utiliser ce type représentation pour le langage SVG. Nous avons défini une structure par type d'élément SVG. Chacune de ces structures utilise un ensemble d'attributs spécifique au type d'élément.

7.2.2.3 Limitation : héritage de propriétés

Bien que cette approche fonctionne pour le langage BIFS, une évaluation de la consommation mémoire sur des contenus SVG du type cartographique nous a permis de nous rendre compte que cette représentation aboutit en général à une surconsommation de mémoire importante, due à un taux faible d'utilisation des attributs, comme nous le détaillerons dans le Tableau 7.1. En réalité, cette représentation ne fonctionne efficacement que sur une certaine catégorie de contenu SVG : les contenus n'exerçant pas le mécanisme d'héritage CSS, comme les contenus conformes au profil LASeR Mini. Pour expliquer ce problème, comparons les exemples de rectangle en BIFS (Code 7.1 et Code 7.2) avec les mêmes exemples exprimés en SVG (Code 7.4 et Code 7.5).

```
<rect width="100" height="100"/>
```

Code 7.4 – Rectangle SVG minimaliste

```
<rect transform="translate(50,50)" fill="red" stroke="blue" stroke-
  width="5" width="100" height="100"/>
```

Code 7.5 – Rectangle SVG détaillé

Ces exemples font apparaître une différence de structuration entre ces langages. Le langage BIFS nécessite beaucoup plus de nœuds pour indiquer l'affichage d'un rectangle (avec toutes ces propriétés d'affichage) que le langage SVG. En effet, le langage SVG opte pour des scènes avec peu d'éléments, mais où chaque élément peut avoir beaucoup d'attributs. Ainsi, la primitive `rect` dans le standard SVG (version 1.2 Tiny) peut potentiellement spécifier 65 attributs décrits dans le Code 7.6. Pour représenter ces fonctionnalités dans le langage BIFS, il faudrait utiliser les nœuds `Switch`, `Transform2D`, `Shape`, `Appearance`, `Material2D` et `XLineproperties`.

```
<rect id="..." xml:id="..." class="..." xml:lang="..." xml:base="..."
  xml:space="..." externalResourcesRequired="..." audio-level="..."
  display="..." image-rendering="..." pointer-events="..." shape-
  rendering="..." text-rendering="..." viewport-fill="..." viewport-fill-
  opacity="..." visibility="..." color="..." color-rendering="..." display-
  align="..." fill="..." fill-opacity="..." fill-rule="..." font-
  family="..." font-size="..." font-style="..." font-weight="..." line-
  increment="..." solid-color="..." solid-opacity="..." stop-color="..."
  stop-opacity="..." stroke="..." stroke-dasharray="..." stroke-
  dashoffset="..." stroke-linecap="..." stroke-linejoin="..." stroke-
  miterlimit="..." stroke-opacity="..." stroke-width="..." text-
  anchor="..." vector-effect="..." focusHighlight="..." focusable="..."
  nav-down="..." nav-down-left="..." nav-down-right="..." nav-left="..."
  nav-next="..." nav-prev="..." nav-right="..." nav-up="..." nav-up-
  left="..." nav-up-right="..." requiredExtensions="..."
  requiredFeatures="..." requiredFonts="..." requiredFormats="..."
  systemLanguage="..." transform="..." x="..." y="..." width="..."
  height="..." rx="..." ry="..."/>
```

Code 7.6 – Attributs possibles pour l'élément rect du langage SVG

Si on analyse plus précisément le Code 7.6, on se rend compte que seuls les attributs en gras s'appliquent réellement et modifient l'aspect visuel du rectangle, et que l'élément `rect` possède un grand nombre d'attributs qui n'affectent pas l'affichage du rectangle, voire qui ne s'appliquent pas du tout au rectangle, comme l'attribut `font-size`. Cette remarque s'applique de même à tous les éléments du langage SVG : ils possèdent des attributs qui ne les affectent pas directement mais qui sont potentiellement spécifiables du fait du mécanisme d'héritage, décrit au chapitre 3. Dans le langage SVG (version Tiny 1.2), on compte 37 propriétés, dont certaines sont de type complexe (liste de valeurs décimales). L'héritage est une fonctionnalité intéressante pour la création de contenu, pour spécifier un style d'affichage (par exemple la même taille de police de caractères) commun à tout un sous-arbre de la scène. Cela permet également de réduire la taille du fichier XML, comme nous l'avons

décrit au chapitre 3. L'inconvénient de cette technique provient du fait que ces propriétés, même si elles sont a priori inutiles pour l'élément en question, peuvent être accédées par des manipulations de l'arbre de scène et doivent donc être stockées, si elles sont spécifiées, au niveau de chaque élément lors de la phase de lecture du contenu. Ainsi, la structuration des objets SVG selon l'approche BIFS ne convient pas dans le cas général car elle aboutit à des tailles d'objets trop importantes.

7.2.2.4 Seconde proposition : structuration compatible avec l'héritage de propriétés

Comme nous venons de la voir, si une implémentation du langage SVG décide d'allouer de l'espace mémoire dès la création d'un élément pour toutes les propriétés qui peuvent être stockées au niveau de cet élément, la taille mémoire d'un élément SVG sera très importante. Nous décrivons ici la seconde solution que nous avons retenue pour notre implémentation.

7.2.2.5 Description

Nous avons envisagé une autre solution qui consiste à allouer dynamiquement l'espace de stockage de la valeur d'un attribut. Cet espace n'est alloué que si l'attribut est spécifié pour cet élément dans le document XML. Cette méthode ne présente vraiment d'intérêt que si une liste des attributs alloués dynamiquement est créée. En effet, dans notre implémentation, utilisant le langage C, l'espace utilisé pour stocker la valeur d'un attribut (couleur, épaisseur de trait, point, booléen) est en général inférieur ou égal à l'espace de stockage d'un pointeur, sauf pour quelques attributs de type complexe (matrice, liste de points, …). Ainsi, s'il faut réserver en mémoire un pointeur pour chaque attribut spécifiable, cela revient quasiment à réserver en mémoire l'espace pour stocker la valeur directement.

Nous avons donc considéré la structure mémoire correspondant à un élément SVG, décrite dans le Code 7.7. Nous utilisons une liste, de taille variable, contenant les attributs réellement spécifiés dans le document. Cette liste est construite lors de la lecture d'un élément et, pour chaque attribut spécifié, une entrée est ajoutée. Cette entrée doit contenir l'identifiant de l'attribut (son nom sous forme d'une chaîne de caractères, ou un identifiant numérique unique dans une version plus optimale et compatible avec un encodage binaire) et l'emplacement mémoire pour stocker la donnée.

```
struct AttributSVG {
   int identifiant_attribut;
   void *valeur;
};
struct ElementSVG {
   int type;
   Liste(AttributSVG) attributs;
};
```

Code 7.7 – Exemple de structure décrivant un élément SVG

Suite à ces choix, nous avons développé un module de lecture de contenu SVG dont le rôle est de créer les structures d'élément SVG, d'allouer les attributs dynamiques et de stocker les valeurs des attributs spécifiés. Ce module est conçu pour s'interfacer les modules classiques de lecture de document XML.

Nous l'avons interfacé notamment avec le logiciel `LibXML`. Nous avons expérimenté les deux interfaces habituelles de lecture XML : DOM et SAX. Cette dernière interface étant plus propice notamment pour la lecture progressive, que nous avons préconisée dans la partie précédente, c'est celle que nous décrivons ici dans le Code 7.8.

```
DébutLectureElement(ElementSVG P) {
  Création d'un élément E à partir du nom issu du fichier XML ou binaire
  Ajout de l'élément comme enfant de l'élément P

  Si l'élément E est une animation ou un élément temporel,
    Différer l'interprétation des attributs

  Pour chaque attribut lu,
    Si l'attribut référence un élément X non résolu,
      Différer l'interprétation
    Sinon
      Allocation de la mémoire nécessaire pour cet attribut
      Stockage de la valeur typée à cette adresse mémoire
      Ajout de l'attribut dans la liste des attributs de l'élément E

  Si l'élément E possède un identifiant,
    Tenter l'interprétation des attributs différés

  Initialisation de l'élément E pour le rendu progressif
}

FinLectureElement(){
   Déclenchement l'évènement "fin de chargement" de l'élément pour terminer
   le rendu progressif
}
```

Code 7.8 – Pseudo-code décrivant un algorithme simplifié de lecture d'un élément SVG

Une structure d'élément E est instanciée en fonction du nom ou de l'identifiant d'élément lu dans le fichier XML ou le fichier binaire. Cet élément E est ajouté dans l'arbre de scène comme enfant de l'élément parent P. Ensuite, la lecture des attributs débute et la valeur de l'attribut est stockée dans une structure décrite dans le Code 7.7. Cependant, la lecture de certains attributs peut nécessiter d'abord la lecture d'autres éléments de la scène. Par exemple, si l'élément en cours de lecture est un élément d'animation, la lecture des attributs décrivant les valeurs d'interpolation ne peut être effectuée que lorsque le type de ces attributs est connu, c'est-à-dire quand l'élément cible de l'animation a été lu. Le même problème se pose pour la lecture des attributs de temps ou pour les attributs référençant un élément (comme l'attribut `xlink:href`). L'interprétation de ces attributs est donc différée, et à chaque nouvel élément lu, qui possède un identifiant, elle est tentée à nouveau.

Cette méthode présente donc l'avantage théorique d'une réduction de la mémoire utilisée mais l'accès à un attribut nécessite le parcours de la liste des attributs spécifiés. Il faut noter que seuls les accès en lecture sont affectés, c'est-à-dire durant la phase de composition. Or, pendant cette phase, tous les attributs de l'élément sont lus, nous avons donc considéré une façon simple de limiter l'augmentation du temps de lecture de ces attributs, en utilisant une structure contenant tous les attributs possibles

dans le langage SVG. Cette structure est créée temporairement quand on souhaite accéder à un ensemble d'attributs d'un même élément. Grâce à cette structure, le parcours des attributs spécifiés n'est effectué qu'une seule fois, pour remplir cette structure.

7.2.2.6 Résultats et limitations

Suite à ce travail d'implémentation, nous avons pu vérifier les différents points suivants :

- Une réduction de la consommation mémoire d'un facteur 4 en moyenne, facteur qui augmente avec la taille du contenu, entre la solution précédente (numérotée 1) et cette solution (numérotée 2). Quelques résultats chiffrés sont indiqués dans le Tableau 7.1. On note également une légère diminution du temps de lecture, vraisemblablement due à une politique d'allocation mémoire plus efficace.

Séquence SVG	Nombre d'éléments	Nombre d'attributs	Consommation Mémoire (Ko)	
			1	2
Plan.svg	287 059	246 296	259 120	56 233
GareDuNord.svg	13 263	50 891	13 903	5 801
Plan_layer_1.svg	12 896	27 659	13 845	5 206
Face_frame_1.svg	490	980	555	228
Cee.svg	81	99	407	306
Butterfly.svg	4	14	92	90

Tableau 7.1 – Réduction de la consommation mémoire

- La rapidité de notre algorithme de lecture par rapport à des implémentations de référence comme le montre le Tableau 7.2. Les séquences ont été modifiées, en plaçant tous les éléments dans un groupe dont la propriété d'affichage `display` indique qu'ils ne doivent pas être rendus, afin de tenter de comparer uniquement les phases de lecture des implémentations.

Séquence SVG	GPAC	ASV 6.0	Opera 9.10	Firefox 2.0.0.1	Renesis 0.2.0
Plan_layer_1.svg	1 s / 8 228 Ko	5 s / 47 100 Ko	5 s / 27 364 Ko	17s / 39 668 Ko	3 s / 28 996 Ko
Plan.svg	7 s / 100 Mo	N/A	24 s / 202 Mo	N/A	5 s / 115 Mo
GareDuNord.svg	1 s / 9 Mo	1 s / 40 Mo	2 s / 18 Mo	4 s / 23 Mo	<1s / 13 Mo

Tableau 7.2 – Comparaison du temps de lecture et de la consommation mémoire pour différentes implémentations de référence

- Une augmentation négligeable du temps de composition d'un élément SVG suite à l'accès aux attributs dynamiques grâce à la structure regroupant 'à plat' tous les attributs possible en SVG. Par exemple, sur la séquence 'plan.svg' qui comporte 246 296 attributs, le temps d'accès à ces attributs durant la phase de composition est mesuré comme représentant 0.1% du temps total d'un cycle de présentation.

7.2.3 Synthèse sur l'efficacité de la lecture de scènes

Dans cette partie, nous avons tout d'abord redémontré, sur des contenus SVG, que l'utilisation d'un lecteur de scènes binaires était profitable notamment pour la réduction de la consommation mémoire et la rapidité de lecture. Nous nous sommes ensuite intéressés aux structures d'arbres de scènes générés en sortie de ce lecteur. Nous avons analysé les langages BIFS et SVG afin de comparer la structuration des primitives du langage. Nous avons souligné les avantages et inconvénients de chaque solution dans le domaine de la création de contenu et celui de la consommation mémoire. Nous avons exploité cette analyse afin de déterminer une structuration efficace des objets mémoires nécessaires à la lecture des scènes SVG. Cette structuration s'appuie sur des listes d'attributs dynamiquement alloués. Elle permet une consommation mémoire optimale, car seuls les attributs spécifiés sont effectivement alloués; et un accès relativement rapide aux données pour la lecture et l'écriture.

7.3 Composition optimisée de scènes SVG

L'étape de composition est l'étape dans le cycle de présentation qui suit la lecture/décodage du contenu. Cette étape est une étape supplémentaire nécessaire par rapport à la visualisation de média classique (vidéo, audio), il est donc intéressant de décrire plus en détail son fonctionnement afin de bien comprendre les enjeux en terme de performance.

7.3.1 Fonctionnement de la phase de composition

Le but de cette étape est de produire une représentation modifiée de la scène, à chaque instant de présentation, à partir de la représentation mémoire de la scène issue du processus de lecture. Le module de composition doit, à chaque rafraîchissement de l'écran, déterminer les paramètres (aspects, formes, positions, …) des différents éléments audiovisuels en fonction des animations, de l'interactivité, éventuellement des mises à jours issues des flux de descriptions de scènes.

Cette étape peut parfois être intégrée au décodage de la scène, dans certains cas, notamment quand la scène est statique (sans animation, sans interactivité) et quand elle n’utilise pas de média externes (image, sons, vidéos …), mais ce n'est pas le cas général. Dans le cas général, le module de composition est un module séparé du module de décodage pour deux raisons :

- Tout d’abord, comme son nom l’indique, il s’agit de composer des données issues de plusieurs sources, de plusieurs décodeurs (vidéo, audio, images fixes, texte), qui ne fonctionnent pas

nécessairement à la même cadence (exemple : un décodeur vidéo à 15 Hz et un décodeur vidéo à 25 Hz).

- Ensuite, même dans le cas d'un contenu ne nécessitant aucun décodeur annexe, la scène pouvant évoluer dans le temps, dans le cas d'animations ou d'interactions utilisateur, il est nécessaire de recomposer la scène alors même que le lecteur/décodeur de scène n'a pas transmis de données supplémentaires.

7.3.2 Rapidité de composition

La composition implique, à chaque cycle de présentation, un parcours de la représentation mémoire de l'arbre de scène et la lecture des attributs des éléments de cet arbre. Pour obtenir une expérience fluide, notamment en cas d'animation et d'interactivité, nous l'avons dit, il faut que les cycles de présentation soient très courts, donc notamment que l'étape de composition soit la plus rapide possible. Nous avons, dans la partie précédente, décrit comme les objets mémoire étaient structurés. Nous nous intéressons ici aux parcours de ces objets et aux traitements associés qui peuvent être également complexes et coûteux en temps.

Pour limiter le temps de la phase de composition, on peut :

- limiter le temps total passé dans la phase de composition, en limitant le nombre de parcours de la scène par seconde mais au détriment de la fluidité;
- ou pour chaque parcours, limiter l'ampleur du parcours, c'est-à-dire la profondeur de la traversée de l'arbre de scène;
- et enfin, limiter les opérations à effectuer au niveau de chaque nœud de l'arbre.

Différents travaux ont été publiés sur la question. Le groupe de travail SVG réfléchit, dans le cadre de la standardisation du profil Full de la version 1.2 du langage SVG, à l'utilisation d'indicateurs des parties statiques de la scène. La version du 27 octobre 2004 [69] mentionne les propriétés `cache` et `static` décrivant respectivement les parties de la scène devant être retracées souvent et celles qui a priori ne seront pas modifiées. D'autres travaux s'orientent vers la détection des parties de la représentation interne qui n'évolue pas d'un cycle à l'autre afin de ne pas les parcourir au cycle suivant. Par exemple, il n'est pas nécessaire de parcourir les parties de la scène non affectées par l'animation ou l'interaction. Cependant, la détection et le suivi de ces zones 'statiques' de la scène n'est pas un processus simple. Il faut notamment mettre en place des caches qui peuvent être coûteux en mémoire. De plus, la détection et le suivi des zones non modifiées nécessitent de nombreux tests qui ralentissent l'exécution. Donc, si la scène est très volumineuse, il peut arriver qu'une recomposition complète soit plus efficace qu'un suivi et une recomposition des changements. Un

compromis doit être trouvé entre capacité mémoire et vitesse d'exécution. On pourra également lire [01] à ce sujet.

Notre démarche s'inscrit également dans cette optique de limiter l'ampleur des parcours et traitements de l'arbre. Pour cela, nous considérons la composition d'une scène SVG comme le traitement de trois types de primitives :

- les primitives temporelles, c'est-à-dire les animations et les éléments audiovisuels;
- les primitives liées aux interactions utilisateur, c'est-à-dire les écouteurs, les gestionnaires d'évènements et les scripts;
- et enfin, les primitives graphiques et celles relatives au positionnement spatial des primitives graphiques.

Toutes ces primitives nécessitent un traitement spécifique durant la phase de composition. Une approche simple de composition consisterait à parcourir l'arbre de scène et à appliquer ce traitement spécifique à chaque élément au moment de la traversée de cet élément. Il est évident que cette approche n'est pas satisfaisante car si la scène contient beaucoup d'objets graphiques et peu d'animations, les animations ayant besoin d'être parcourues régulièrement pour atteindre une fluidité satisfaisante, les primitives graphiques seraient alors inutilement parcourues.

Cependant, il n'est pas immédiat que, dans toutes les scènes, du fait de mécanismes d'héritage et d'interactivité, les primitives temporelles, d'interactivité et graphiques peuvent être traitées séparément. Nous analysons ici ces trois catégories de primitives afin de montrer qu'il est possible de séparer le traitement entier de l'arbre en plusieurs traitements disjoints de ces trois catégories pour obtenir un algorithme de composition des scènes SVG plus rapide que l'approche simple précédente.

7.3.2.1 Traitement des éléments temporels

Pour les éléments temporels, le traitement effectué dans la phase de composition peut se dérouler comme suit :

- Déterminer si un élément temporel est actif, par la construction des intervalles d'activité à partir des valeurs temporelles de début et de fin d'activation, et générer les évènements correspondants (début, fin, répétition).
- Dans le cas où un intervalle d'activité existe :
    - Calculer la fraction de temps écoulé dans la durée simple depuis le début de cette période d'activité et le nombre de répétitions qui ont eu lieu.
    - Appliquer un traitement associé. Dans le cas d'un élément d'animation, ce traitement est le calcul de la valeur d'interpolation et la modification de la scène. Dans le cas

d'un élément vidéo, il consiste à afficher l'image précise correspondant à l'instant de composition.

La première partie est une tâche relativement complexe. L'activité d'un élément peut en effet dépendre d'évènements qui se produisent dans d'autres parties de l'arbre de scène, comme dans l'exemple du Code 7.9.

```
<A begin="B.begin" …>
…
<B begin="3; A.end" …>
```

Code 7.9 – Démarrage conjoint de deux éléments temporels

Dans une approche basique, si on décide de traiter l'activation des éléments temporels en fonction de l'ordre dans lequel ils sont déclarés dans l'arbre de scène, A ne sera pas démarré pendant le même cycle de composition que B. Cette approche pourrait résulter en une désynchronisation des animations A et B, potentiellement non acceptable. Du fait de ces dépendances, il faut donc, à chaque activation, désactivation ou répétition d'un élément temporel, notifier les autres éléments temporels de ce changement d'état puis parcourir à nouveau la liste des éléments temporels, jusqu'à arriver soit à un état stable (plus de notifications), soit à détecter des références cycliques correspondant à un contenu erroné.

De plus, il est intéressant de remarquer que, si tous les éléments de l'arbre de scène partagent la même base de temps, comme c'est le cas dans un document SVG, la résolution des dépendances temporelles entre ces éléments devient donc indépendante de leurs positions dans l'arbre de scène.

On retiendra donc que notre algorithme exploite cette propriété afin de ne pas effectuer inutilement le traitement associé à un élément temporel actif. Ainsi dans notre algorithme, la phase de traitement d'un élément temporel actif (interpolation ou affichage) est effectuée après le traitement d'activation/désactivation/répétition de tous les éléments temporels de la scène. La conséquence, dans le cas d'une scène SVG large et comportant de nombreux éléments mais très peu d'éléments temporels, est que le temps passé pour la résolution des dépendances temporelles est réduit car seul un parcours limité de l'arbre est effectué.

Les paragraphes suivants précisent à quel moment est effectué le traitement spécifique à chaque élément temporel actif :

- D'après le modèle SMIL d'animation, décrit dans le chapitre 2, le résultat d'une animation requiert la résolution des attributs, de la cible de l'animation, dont la valeur est `inherit`. Cette résolution nécessite de connaître la valeur résolue du même attribut au niveau de l'élément parent de cette cible. Cependant, cette valeur est, à son tour, le résultat des animations à ce niveau de l'arbre, qui découle des animations au niveau parent. Ce comportement implique que la valeur issue d'une animation dépend de la position de la cible

de l'animation dans l'arbre de scène. Ainsi, il n'est pas possible de séparer le traitement des animations actives du traitement de l'arbre en entier. En d'autres termes, pour traiter une animation, il faut parcourir l'arbre de la racine jusqu'à la cible de l'animation.

- A l'inverse des éléments d'animation, le traitement associé à un élément média, quand il est actif, consiste à mettre à jour l'image ou les échantillons audio issus du décodeur vidéo ou audio. Ce traitement ne dépend pas de la position de l'élément média dans l'arbre. Nous complétons donc notre algorithme précédent en utilisant une liste des éléments médias, au niveau du module de composition, parcourue à chaque cycle de composition afin de mettre à jour les données issues des décodeurs. L'avantage de cette liste est de permettre la mise à jour des données média indépendamment de la traversée de l'arbre. Ainsi, dans le cas d'une scène statique, composée de nombreux éléments graphiques mais ne comportant qu'un unique élément vidéo, il ne sera pas nécessaire de parcourir l'arbre en entier pour mettre à jour l'image issue du décodeur vidéo.

7.3.2.2 Traitement des éléments liés au modèle évènementiel DOM

Comme nous l'avons décrit dans le chapitre 4, le modèle évènementiel DOM, utilisé dans les scènes SVG, se base sur la déclaration d'écouteurs d'évènements, sur des mécanismes de propagation d'évènements et sur l'utilisation de gestionnaires d'évènements. Or, le comportement des écouteurs et gestionnaires d'évènements (script, animation ou hyperliens) ne dépend pas de la position de cet élément dans l'arbre. Seule la position de l'élément cible de l'évènement a de l'importance. En effet, du fait des propagations ascendantes et descendantes (bouillonnement et capture), on ne peut pas ignorer la position de cet élément car un évènement déclenché à un endroit dans l'arbre peut être traité à un niveau supérieur dans l'arbre.

De manière précise, le traitement d'un évènement implique :

- le parcours, pour chaque élément cible d'évènement, de l'arbre de scène en entier pour déterminer le chemin de la racine jusqu'à cet élément. Ce parcours, nécessaire pour la phase de capture, est effectué une seule fois, pour toute la phase de composition, à la mise en place de l'écouteur d'évènement;
- la détermination de l'élément cible de l'évènement selon la position du pointeur ou l'élément ayant le focus pour les évènements issus du clavier;
- le parcours de ce chemin de la racine jusqu'à l'élément cible, à chaque fois qu'un évènement est émis et pour lequel un écouteur est en place;
- et le parcours en remontant ce chemin de l'élément cible jusqu'à la racine pour les évènements supportant le mode bouillonnement.

On remarque donc qu'il peut y avoir deux parcours, potentiellement coûteux, à effectuer à chaque fois qu'un évènement est émis. Dans le cas où la méthode de capture n'est pas supportée, c'est le cas dans le profil Tiny de la norme SVG, les deux premiers parcours deviennent inutiles. Il suffit de déterminer les cibles des évènements, de propager les évènements à toutes ces cibles et enfin de remonter l'arbre de scène à partir de ces différentes cibles à chaque fois qu'un évènement est émis. De plus, si les écouteurs limitent la propagation à la phase de ciblage (`propagate='stop'`), le traitement des évènements sera encore plus rapide car cette dernière remontée ne sera pas effectuée.

Ainsi, le traitement des évènements peut être rendu indépendant de la position des différents éléments dans l'arbre de scène sous les conditions suivantes :

- maintien, au niveau du module de composition, de la liste des éléments cibles d'évènements;
- maintien, au niveau de chaque élément cible, des écouteurs d'évènements associés;
- et maintien, au niveau chaque élément de l'arbre de scène, d'un lien vers l'élément parent.

De plus, du fait que les zones sensibles à la souris découlent du positionnement des objets, et du fait du caractère animable de la propriété `pointer-events` qui détermine si un élément est sensible aux évènements souris, notre algorithme effectue le traitement des évènements après la traversée de l'arbre.

7.3.2.3 Traitement des autres éléments

Nous nous intéressons ici aux traitements associés aux éléments autres que les éléments d'animation et ceux d'interaction. Ces éléments sont les éléments de la description de la scène qui ont pour but le positionnement spatial et l'affichage des éléments visuels. La phase de composition pour ces éléments consiste à déterminer, pour un temps de scène donné, les paramètres d'affichage des éléments visibles (couleur, taille, position, facteur d'échelle, épaisseur de trait, …) et à construire la liste d'affichage qui permettra ensuite à la phase de rendu de construire et d'afficher l'image finale.

Bien que seuls les éléments visibles situés dans les feuilles de l'arbre de scène (dans lequel les éléments d'animation et les éléments d'interactivité ont été retirés) produisent un résultat visuel, il est néanmoins indispensable de parcourir l'arbre de scène en entier pour produire la liste d'affichage. En effet, d'une part les paramètres de positionnement (rotation, translation) sont déduits de la composition matricielle des transformations successives rencontrées entre la racine de l'arbre et l'élément visuel; et d'autre part, les paramètres d'affichage (épaisseurs de traits, couleurs …) sont eux déduits de l'animation et de l'héritage des propriétés de style entre la racine de l'arbre et l'élément concerné. De ce fait, il est impossible de dissocier le traitement des éléments graphiques de celui de la traversée de l'arbre. Nous verrons, dans la section 7.4, que nous nous attacherons donc à minimiser le traitement des éléments graphiques.

7.3.2.4 Synthèse

En résumé, nous proposons un algorithme de composition de scènes SVG qui se base sur la réduction de l'ampleur des parcours de l'arbre de scène pour le traitement des éléments temporels et des éléments liés au modèle évènementiel. A chaque étape de composition, il se découpe en quatre phases :

- La première phase consiste à déclencher et à traiter les évènements d'activation, désactivation, répétition des éléments temporels, en effectuant uniquement le parcours de ces derniers et non pas le parcours complet de l'arbre.
- La seconde phase consiste à mettre à jour les éléments audiovisuels activés à la phase précédente.
- La troisième phase consiste à traverser un arbre de scène réduit, ne comportant que les éléments de positionnement spatial, les éléments graphiques et les éléments audiovisuels, pour appliquer de manière conjointe les animations et l'héritage et construire la liste d'affichage.
- Enfin, la dernière phase consiste à traiter les évènements utilisateurs et les évènements systèmes en parcourant uniquement la liste des éléments cibles pour déterminer si un évènement doit être utilisé et en exécutant le gestionnaire d'évènement associé le cas échéant. A nouveau, grâce au maintien de la liste des éléments cibles au niveau du module de composition, cette étape ne nécessite pas le parcours complet de l'arbre et peut être très efficace si seule la phase de ciblage est utilisée.

7.3.3 Composition et consommation mémoire

Au niveau de la consommation mémoire, l'algorithme que nous proposons implique, de manière générale, une consommation mémoire plus importante du fait des points suivants :

- du stockage, au niveau du module de composition, de :
  - la liste des éléments temporels afin de déterminer les activations/désactivations sans recourir à un parcours complet de l'arbre,
  - la liste des éléments média pour mettre à jour les données issues des décodeurs,
  - et la liste des cibles d'évènement pour le traitement des évènements indépendamment de la traversée de l'arbre des éléments graphiques.
- du stockage, dans la structure d'élément décrite précédemment, en plus d'une liste des enfants qui appartiennent à un élément, de :
  - la liste des animations qui s'appliquent à cet élément,
  - un pointeur vers l'élément parent afin de propager les évènements en cas de bouillonnement et de capture,

    - et de la liste d'écouteurs pour lesquels l'élément est ciblé.

- et de l'application conjointe des mécanismes d'héritage et d'animation.

Les deux premiers points sont simples à comprendre. En revanche, le dernier point mérite plus d'explications.

Comme nous l'avons décrit dans les chapitres précédents, les descriptions de scènes au format SVG s'appuient sur les normes SMIL pour l'animation et CSS pour les notions d'héritage. L'implémentation conjointe de ces deux normes implique notamment que le lecteur de scènes multimédia doit supporter les concepts de :

- valeur de base : déduite de la valeur spécifiée dans le document SVG ou de la valeur par défaut définie par le langage;
- valeur calculée : résultat de l'héritage et de la cascade CSS;
- valeur de présentation : résultat conjoint de l'animation et de l'héritage.

Si ces valeurs doivent effectivement être déterminées ou calculées pour composer une scène SVG, les conserver, toutes les trois en mémoire, à chaque instant, pour chaque attribut et chaque élément de la scène conduit à une surconsommation mémoire trop importante. Dans cette partie, nous proposons une méthode pour présenter une scène SVG faisant bien appel à ces trois notions (valeur de base, valeur calculée, valeur de présentation) tout en conservant une occupation mémoire minimale.

Nous partons des constats suivants :

- Seuls les attributs susceptibles d'être animés doivent conserver une séparation entre valeur de base et valeur d'animation. Plus précisément, seuls les attributs réellement animés dans une scène ont besoin de conserver la valeur de base, et ce, seulement pour le cas où l'animation restaurerait cette valeur à la fin de la période d'activité.
- Seuls les attributs qui représentent des attributs héritables ou propriétés doivent utiliser les notions de valeur spécifiée et valeur calculée.

Sur la base de ces constats, notre proposition repose sur trois principes :

- L'utilisation d'un unique espace mémoire par attribut au niveau de l'élément cible de l'animation. La représentation mémoire décrite dans le Code 7.7 n'est donc pas modifiée.
- L'allocation de l'espace mémoire pour la valeur de base au niveau de l'élément d'animation. Ainsi, la mémoire supplémentaire pour la valeur de base d'un attribut n'est allouée que si une animation cible effectivement cet attribut.
- L'utilisation d'un contexte de propriété CSS, stockée dans la pile d'exécution du module de composition, transmis lors de la traversée de l'arbre de scène, potentiellement modifié au

niveau de chaque élément de l'arbre, par l'héritage; et rétabli à la fin du traitement de l'élément. Ainsi, le stockage des valeurs des propriétés calculées CSS n'est que temporaire, et lors de la composition d'un élément de profondeur *p* dans un arbre composé de *N* éléments, il n'y aura que *p+1* contextes de propriétés sauvegardées et non *N*.

Cette proposition aboutit donc à l'algorithme simplifié de composition d'un élément SVG donné dans le Code 7.10.

```
TraverseElement(Element E, ContexteProprietes C) {
 Si (E == racine et première composition)
  Mettre en place dans C les valeurs initiales des propriétés

 Sauvegarder les propriétés du contexte C

 Pour toutes les propriétés P, appliquer l'héritage:
  Si E.P != inherit alors pointer la valeur P de l'élément E dans C

 Pour toutes les animations A dont la cible est E, dans l'ordre
  d'activation,
     S'il s'agit de la première évaluation, stocker la valeur de base
     Si les valeurs clés de A utilisent inherit, résoudre à l'aide de C
     Si A est terminée, écraser la valeur de l'attribut dans E par
                                la valeur de base stockée dans A
     Sinon calculer la valeur interpolée et écraser la valeur de l'attribut
                                                              dans E

  Pour tous les enfants E' de E, appeler TraverseElement(E', C)
  Rétablir les valeurs sauvegardées des propriétés dans C
}
```

Code 7.10 – Algorithme simplifié de composition d'un élément SVG

Nous présentons ici une synthèse des propositions précédentes sous la forme d'un algorithme complet de composition de scènes SVG. L'algorithme final obtenu est décrit dans le Code 7.11. Nous proposons ensuite d'analyser les limitations de cet algorithme.

```
CycleDeComposition() {
   Tant qu'un état stable n'est pas trouvé,
     Parcourir la liste des éléments temporels,
     Evaluer les intervalles d'activité,
     Transmettre les évènements de début/fin/répétition d'élément.

   Pour chaque évènement reçu depuis le précédent cycle,
     Pour chaque écouteur dans la liste des écouteurs,
       Si l'évènement correspond,
         Activer le gestionnaire d'évènements
         Si la propagation continue, faire bouillonner l'évènement

   Parcourir la liste des éléments média,
      Mettre à jour les mémoires de composition issues des décodeurs média

   TraverseElement(racine);
}
```

Code 7.11 – Algorithme simplifié pour la composition d'une scène SVG

7.3.4 Résultats et limitations

Notre algorithme possède deux limitations que nous exposons ici : la première liée aux interfaces de manipulation des arbres de scènes SVG et la seconde liée à des cas particuliers d'imbrication héritage/animation.

7.3.4.1 Limitation dans la manipulation de l'arbre de scène

Comme nous l'avons indiqué auparavant, deux ensembles d'interfaces sont actuellement disponibles dans le langage SVG : DOM ou Micro DOM. Nous présentons ici notre analyse de la complexité de ces accès, les limitations de notre proposition dans ce domaine ainsi que les raisons pour lesquelles notre proposition reste toujours satisfaisante.

7.3.4.2 L'interface DOM

Un arbre DOM est, de manière simplifiée, un arbre composé de nœuds correspondant aux éléments présents dans un fichier XML. De plus, chaque attribut spécifié dans le document XML correspond à une structure d'attribut DOM, associé à ce nœud, et dont la valeur est une chaîne de caractère. S'il n'existe pas de grammaire attachée au document, les attributs non spécifiés dans le document XML ne sont pas présents dans la structure de nœud DOM. En revanche, si le document XML possède une grammaire associée sous la forme d'une DTD, et que celle-ci définit une valeur par défaut pour un attribut, une structure d'attribut DOM est créée avec cette valeur par défaut, et cette structure est attachée au nœud DOM en question.

Ces particularités de l'interface DOM posent des problèmes :

- Le premier problème est l'utilisation de chaînes de caractères. On peut décider de stocker les valeurs sous forme typée (nombre entier, booléen, nombre décimal, triplet de couleurs), comme c'est le cas dans notre proposition, mais à chaque accès à un attribut, le module de composition devra convertir la chaîne en représentation typée ou inversement. Ainsi, en réduisant la consommation mémoire grâce à l'utilisation de valeurs typées, on réduit également la rapidité d'accès aux données.
- Le second problème concerne l'accès aux attributs quand ils ne sont pas spécifiés dans le document. L'interface DOM indique, dans ce cas, que la fonction `getAttribute` doit retourner une chaîne de caractère vide (si aucune grammaire n'est associée). Cela oblige une implémentation à maintenir d'une part le fait qu'une grammaire soit associée et le fait qu'un attribut ait été spécifié ou non.

Notre proposition n'est donc pas optimale pour un interfaçage avec l'API DOM. Cependant, l'interface DOM est remplacée pour les raisons exposées précédemment, dans le profil Tiny de la version 1.2 du langage SVG, par une interface de programmation plus simple qui ne pose pas ces problèmes et pour

laquelle il est plus pertinent d'évaluer notre algorithme de composition. Cette interface s'appelle MicroDOM.

7.3.4.3 L'interface MicroDOM

L'interface MicroDOM définit la notion de *Trait* pour accéder aux attributs en s'appuyant sur deux points :

- Tout d'abord, la spécification offre la possibilité d'accéder aux attributs en écriture et en lecture par des fonctions typées. De ce fait, notre choix de ne pas utiliser une représentation des attributs sous forme de chaîne de caractères pour les valeurs des attributs est compatible.
- Ensuite, les attributs non spécifiés dans le fichier SVG sont représentés dans l'arbre MicroDOM par une structure dont la valeur est effectivement la valeur par défaut définie dans la spécification SVG. Il n'y a pas de distinction entre valeur par défaut et valeur spécifiée. Sur ce point, l'arbre MicroDOM diffère de ce que nous proposons, car nous n'allouons en mémoire que les attributs spécifiés. Cependant, il est possible d'étendre notre implémentation pour retourner une valeur par défaut à une requête d'accès sur un attribut non spécifié.

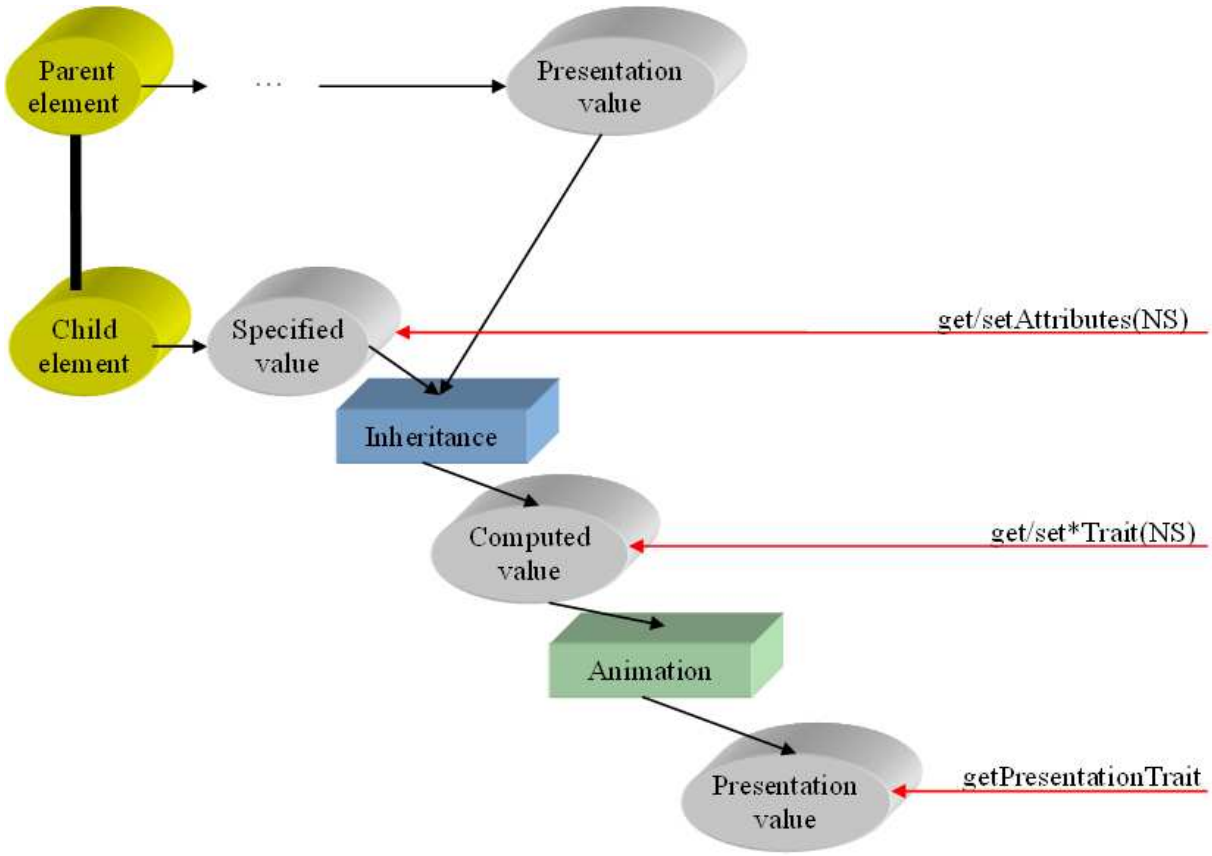


Figure 7.3 – Fonctions d'accès aux attributs DOM et aux Traits MicroDOM

Notre structuration des éléments et notre algorithme de composition sont donc compatibles avec l'interface MicroDOM sur ces deux points. Cependant, certains problèmes subsistent. En effet, la spécification SVG indique deux points importants. Elle indique que la valeur retournée par les fonctions `getTrait` est la valeur de base en cas d'animation, c'est-à-dire la valeur après héritage potentiel, autrement dit la valeur calculée selon la norme CSS. Elle indique également que la valeur retournée par la fonction `getPresentationTrait` est la valeur de présentation si la valeur est animée. Les différentes valeurs retournées par ces différentes fonctions sont illustrées dans la Figure 7.3.

Les problèmes liés à notre algorithme sont donc les suivants :

- Dans le cadre de notre proposition, nous avons opté pour un arbre de scène dont les attributs stockent la valeur de présentation, ou la valeur spécifiée s'il n'y a pas d'animation. Ainsi la valeur après héritage n'est calculée que temporairement pendant la phase de composition d'un élément (comme nous l'avons décrit précédemment) et nous n'avons donc pas duplication de la mémoire nécessaire par attribut. Il n'est donc pas possible d'obtenir la valeur calculée selon CSS sans effectuer une phase de composition supplémentaire de la racine jusqu'à l'élément en question.
- De même, dans notre proposition, pour obtenir la valeur spécifiée dans le document XML, il faut déterminer si une animation s'applique et si oui, obtenir la valeur de base à partir de la valeur sauvegardée dans la structure d'animation. Ceci suppose donc un accès, possible, mais plus lent à la valeur de base.

Notre proposition présente donc l'inconvénient de ne pas offrir une réponse rapide, dans certains cas, aux appels de fonctions `getTrait`. Ceci est le résultat du fait que notre implémentation privilégie dans ce cas la consommation mémoire à la rapidité d'accès mais ne constitue pas une incompatibilité avec le modèle MicroDOM.

7.3.4.4 Limitation lié à l'héritage

Une seconde limitation de notre proposition est toujours liée à la norme CSS et à la notion d'héritage. L'héritage a toujours lieu, selon la norme CSS, en descendant l'arbre. Cependant, SVG autorise des éléments à référencer d'autres éléments de l'arbre pour les réutiliser. C'est le cas, par exemple, de l'attribut `stroke` qui peut pointer vers un élément de type `linearGradient`. Dans ce cas, le gradient hérite ses propriétés du parent de l'élément gradient et non du parent de l'élément qui le référence.

Dans notre proposition d'algorithme de composition, nous ne traversons que les éléments graphiques visibles, pour limiter la quantité d'éléments traversés dans l'arbre. Dans certains cas d'utilisation de gradient, quand le gradient est défini dans une partie non visible de l'arbre de scène, et que ses

propriétés sont héritées, notre proposition conduit à forcer une traversée de l'arbre supplémentaire jusqu'à ce gradient, pour déterminer les valeurs correctes de ces propriétés. Ce cas étant peu fréquent, nous considérons cette limitation comme acceptable.

7.4 Rendu optimisé de scènes SVG

Le rôle de la phase de rendu est, comme son nom l'indique, de procéder au rendu visuel des objets graphiques (images et dessins synthétiques) de la liste d'affichage et au rendu sonore des objets audio. Nous nous intéressons ici uniquement à la partie visuelle du rendu.

Le résultat de la phase de composition, présentée précédemment, est la liste d'affichage (*display list*), c'est-à-dire la liste d'objets à afficher, dont les propriétés visuelles sont déterminées et fixes, car les modifications liées aux actions utilisateurs, aux animations, à l'héritage, et aux calculs de positionnement ont été appliquées. Nous présentons, dans cette partie, les principes du rendu d'objets graphiques à partir de cette liste d'affichage et nos propositions dans ce domaine.

7.4.1 Principes du rendu efficace de scènes

Afin d'obtenir une fluidité dans l'affichage graphique, les algorithmes de rendu existants [34] tentent de limiter les accès à la mémoire vidéo. Pour cela, un algorithme de rendu doit déterminer les zones de l'écran qui ont été modifiées. Les algorithmes sont donc amenés à comparer le contenu de la liste d'affichage d'une phase de rendu avec le contenu de la liste d'affichage de la phase de rendu précédente.

Il est important de remarquer qu'afin de pouvoir mettre en place ces algorithmes, le rendu des objets graphiques ne peut avoir lieu qu'après la phase de composition. En effet, il faut que tous les objets aient leurs paramètres d'affichage et de positions déterminés pour pouvoir savoir s'ils ont été modifiés. Par conséquent, pour procéder à un rendu efficace, il faut effectuer un parcours supplémentaire de la scène. Afin d'accélérer ce parcours, lors de la phase de composition, le module de composition enregistre chaque élément graphique visible au niveau du module de rendu dans une liste d'affichage (voir Figure 7.4). Le processus de rendu ne fonctionne donc pas en parcourant l'arbre de scène en entier, seul le parcours de cette liste d'affichage est nécessaire. De cette manière, le rendu d'une scène complexe, composée de nombreux objets est accéléré si très peu d'objets graphiques sont présents ou si très peu d'objets graphiques sont visibles.

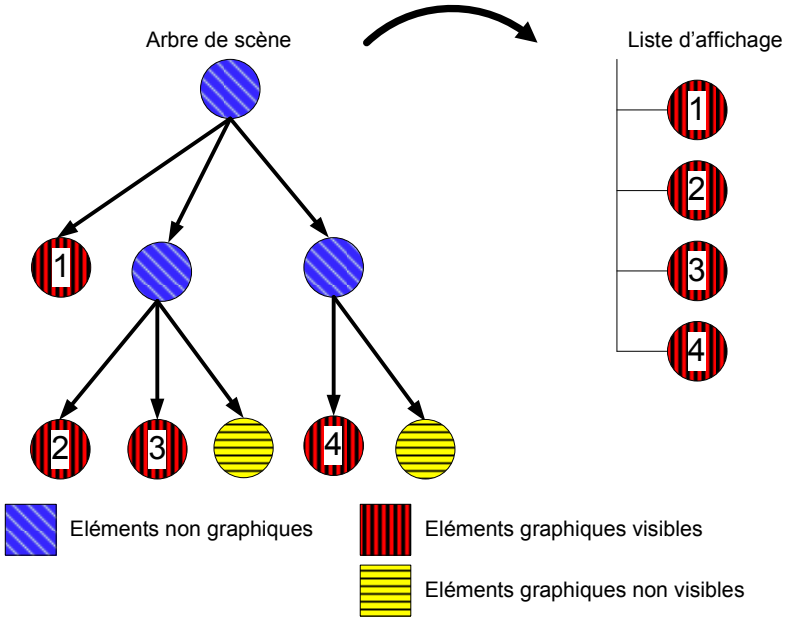


Figure 7.4 – Passage d'un arbre de scène à une liste d'affichage

A chaque cycle de présentation, pour chaque emplacement dans cette liste, trois critères sont comparés en posant les questions suivantes :

- La forme géométrique de l'objet a-t-elle changé ?
- L'apparence (couleur de remplissage, de trait …) a-t-elle changé ?
- La position de l'objet a-t-elle changé ?

Si la réponse est oui à l'une de ces questions, il faut rafraîchir une partie de l'écran. Si l'objet n'a changé que d'apparence, on retracera uniquement la zone délimitée par la boîte rectangulaire englobant la forme géométrique. Si, par contre, l'objet a changé de forme ou s'est déplacé, il faudra retracer l'union de la boîte rectangulaire englobant l'objet de la phase précédente et de celle de l'objet courant.

La description que nous venons de donner des algorithmes de rendu ainsi que la partie purement graphique du rendu qui consiste à réaliser le tracé de courbe de Bézier ou l'affichage de texture (image) sont généralement indépendants du langage de descriptions de scènes. En revanche, l'opération de comparaison des objets présents dans la liste d'affichage peut être accélérée selon le langage de descriptions de scènes utilisé. Le langage Flash est à cet égard le plus avancé. En effet, la structure de la scène est nativement sous forme de liste d'affichage. De plus, le langage ne permet pas de modifier (ni par script, ni par mise à jour, ni par interpolation) la structure interne d'un élément présent dans la liste d'affichage. Ainsi, il est facile de détecter qu'un objet visible est modifié dès que la couche correspondante dans la liste d'affichage est modifiée.

Nous proposons, dans la suite de cette partie, une méthode pour la gestion de la détection des parties modifiées pour des scènes SVG, mais avant cela nous présentons les problèmes spécifiques à ce langage.

7.4.2 Rendu SVG et héritage

Une difficulté pour la gestion et le suivi des objets visibles modifiés dans une scène SVG provient de l'utilisation d'une structure d'arbre de scène arbitraire, où l'arbre de scène n'est pas composé, à l'inverse de l'arbre Flash, d'un dictionnaire et d'une liste d'affichage. Il faut donc surveiller l'arbre en entier pour déterminer si la liste des objets visibles a changé. En effet, de nouveaux objets visibles peuvent apparaître à n'importe quel endroit dans l'arbre. Ce problème, non spécifique au langage SVG (cela s'applique également au langage BIFS), est aggravé du fait de l'utilisation des mécanismes d'héritage. Afin d'illustrer ce problème, comparons le fonctionnement des animations entre le langage VRML et le langage SVG.

Dans le standard VRML, l'animation ne peut avoir lieu que sur un nœud précis identifié par une route reliant un interpolateur à la cible de l'animation. Il est donc facile de détecter une modification sur ce nœud, de déterminer la propriété du nœud qui est changée, et consécutivement de mettre à jour, s'il s'agit d'un nœud visible, la partie de la liste d'affichage correspondante.

Dans le langage SVG en revanche, du fait du mécanisme d'héritage de propriété, une animation peut modifier une propriété à un niveau dans l'arbre de scène et avoir un impact sur tout le sous-arbre ayant pour racine la cible de l'animation. Plus précisément, la partie de l'arbre impactée par une animation est constituée de l'ensemble des éléments pour lesquels la propriété animée est héritée : explicitement (attribut spécifié avec la valeur `inherit`), ou implicitement (attribut non spécifié), comme illustrée sur la Figure 7.5 . La détection des modifications d'affichage en SVG est donc fortement liée à la gestion de l'héritage.

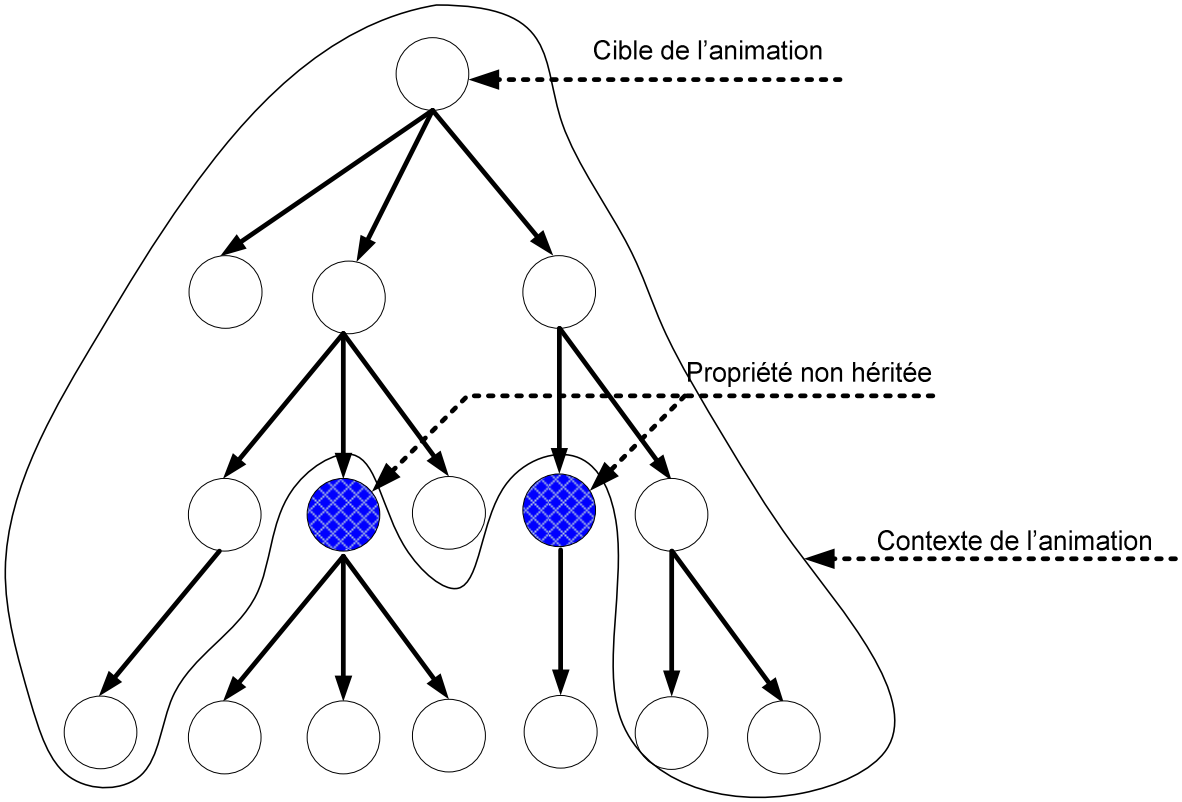


Figure 7.5 – Contexte d'animation en cas d'héritage de propriété

Une autre difficulté est liée à la réutilisation d'éléments SVG. Nous avons expliqué, dans le chapitre 3, qu'il est possible en VRML, grâce à l'attribut `use`, ou en SVG, grâce à l'élément `use`, d'indiquer qu'un élément (et le sous-arbre dont il est la racine) est réutilisé plusieurs fois dans la scène. Cependant, une différence importante entre les deux mécanismes provient à nouveau de la notion d'héritage. En effet, certaines propriétés héritables peuvent modifier l'apparence d'une forme géométrique (`fill`, `stroke`) ou d'autres peuvent modifier la taille du texte (`font-size`, `font-style`). Ainsi, un même élément réutilisé à différents endroits dans un arbre de scène peut avoir un aspect visuel différent. On est donc amené à considérer, dans la liste d'affichage, plusieurs clones d'un élément visible, en fonction de l'endroit dans l'arbre où il est utilisé. De plus, comme une propriété affecte non pas un seul élément mais tout un sous-arbre, la réutilisation d'un élément, en réalité, implique le clonage de tous les éléments visibles du sous-arbre dont il est la racine.

7.4.3 Proposition pour le suivi des objets modifiés en SVG

Comme nous l'avons vu dans les chapitres précédents, les sources de modification dans une scène sont les animations et l'interactivité. Afin de détecter les modifications qui ont lieu sur une scène entre deux phases de rendu consécutives, notre algorithme de rendu s'appuie premièrement sur un suivi des éléments d'animation actifs et des scripts actifs. Pour chaque attribut animé, nous déterminons si le résultat cumulé des animations, tel que décrit dans le chapitre 3, a produit une valeur différente par

rapport au cycle précédent. Si c'est le cas, nous marquons l'élément concerné comme modifié. De même, à chaque exécution de script, si un élément est modifié, nous marquons l'élément.

La difficulté réside dans la façon de marquer le nœud. En effet, nous avons vu que, pour des raisons d'efficacité, la détection des modifications de liste d'affichage s'appuient sur trois critères de comparaisons : forme géométrique, apparence et position. Par exemple, si seule la couleur de remplissage a été modifiée, il ne faut pas recalculer la géométrie de l'objet ou les propriétés de contour car ces opérations peuvent être coûteuses. Nous utilisons donc un marqueur pour déterminer si la géométrie de l'élément graphique est modifiée (`GEOMETRY_DIRTY`). C'est par exemple le cas si on anime l'attribut `width` d'un rectangle ou si on modifie la liste des points composant un polygone. Pour ce qui est de la position, nous effectuons directement la comparaison entre les matrices de positionnement et n'utilisons pas de marqueur. En revanche, pour suivre les modifications d'apparence, l'utilisation d'un marqueur n'est pas suffisante dans le langage SVG du fait de l'héritage de propriétés. En effet, il faut pouvoir distinguer quel paramètre de l'apparence a été modifié pour retracer un minimum d'objets.

Notre algorithme considère donc les marqueurs de modification d'apparence suivants :

```
COLOR_DIRTY, DISPLAYALIGN_DIRTY, FILL_DIRTY, FILLOPACITY_DIRTY,
  FILLRULE_DIRTY, FONTFAMILY_DIRTY, FONTSIZE_DIRTY, FONTSTYLE_DIRTY,
  FONTVARIANT_DIRTY, FONTWEIGHT_DIRTY, LINEINCREMENT_DIRTY, OPACITY_DIRTY,
  SOLIDCOLOR_DIRTY, SOLIDOPACITY_DIRTY, STOPCOLOR_DIRTY, STOPOPACITY_DIRTY,
  STROKE_DIRTY, STROKEDASHARRAY_DIRTY, STROKEDASHOFFSET_DIRTY,
  STROKELINECAP_DIRTY, STROKELINEJOIN_DIRTY, STROKEMITERLIMIT_DIRTY,
  STROKEOPACITY_DIRTY, STROKEWIDTH_DIRTY, TEXTALIGN_DIRTY,
  TEXTANCHOR_DIRTY, VECTOREFFECT_DIRTY.
```

Nous utilisons un marqueur spécifique par propriété. Ces marqueurs sont hérités le long de l'arbre DOM de la même manière que la propriété correspondante. Par exemple, si un élément de l'arbre DOM est marqué avec le marqueur `STROKEWIDTH_DIRTY`, cela signifie que la propriété `stroke-width` a été animée ou modifiée par script sur cet élément ou sur un élément parent. Si les enfants de cet élément utilisent la valeur `inherit` pour cette propriété, ces enfants seront à leur tour marqués avec le marqueur `STROKEWIDTH_DIRTY`. En revanche, si un enfant spécifie pour cette propriété une valeur différente de `inherit`, le marqueur ne sera pas hérité et l'objet ne sera pas rafraîchi.

7.4.4 Résultats et limitations

Nous avons implémenté notre algorithme dans le lecteur GPAC, ce qui nous a permis de valider le fait que le suivi des objets modifiés permet de limiter la taille des zones de l'écran à rafraîchir et donc permet une fréquence de rafraîchissement plus élevée. Par exemple, sur un contenu graphique décrit par un groupe composé d'un rectangle et de 72 cercles, où la propriété de couleur de remplissage est animée au niveau du groupe et où seul le rectangle hérite cette propriété, nous avons mesuré une réduction d'un facteur 9 du temps de rendu entre deux versions du moteur de rendu avec et sans notre

proposition. Cependant, il faut noter que la réduction du temps de rendu dépend fortement du contenu et de la manière dont le contenu est créé : du nombre et de la répartition en groupe des objets animés et non animés, ou de la superficie de la fenêtre de visualisation occupée par ces objets.

Notre implémentation présente actuellement deux limitations. La première concerne la quantité de marqueurs. Celle-ci est directement liée au nombre de propriétés possibles sur un élément. Ainsi, dans un contexte de descriptions de scènes mélangeant plusieurs langages comme XHTML, SVG, CSS, où de nombreuses propriétés sont utilisées, le nombre de marqueurs, et donc de tests à chaque phase de rendu, seront très grands.

La seconde limitation concerne l'affichage de texte. Nos marqueurs sont distincts pour le suivi de l'apparence et de la géométrie de l'objet graphique à tracer. Cependant pour le texte, certaines propriétés de style comme `font-family` (qui modifie la police de caractères utilisée) ou `font-style` (qui indique si le texte est affiché en gras ou en italique) nécessite effectivement de recalculer la forme géométrique du texte. Ces différents marqueurs pourraient donc être fusionnés.

7.5 Conclusion

Dans le cadre de ces travaux, nous avons réalisé une implémentation logicielle des différents algorithmes présentés dans ce chapitre. Cette implémentation s'intègre dans la plateforme logicielle libre nommée GPAC. Au sein de cette plateforme, nous avons donc développé un module de lecture de fichier SVG ainsi qu'un module de décodage de flux binaire au format LASeR, dont le rôle est de créer les mêmes structures d'arbre de scène que le lecteur de fichiers XML. Cette implémentation a servi, dans le cadre des activités de normalisation SVG et LASeR, à montrer la faisabilité d'une implémentation se basant sur un seul arbre de scène capable de représenter des données issues d'un flux LASeR ou d'un document SVG. La preuve de cette faisabilité a permis au consortium 3GPP de baser sa solution technique pour les scènes multimédia interactives et dynamiques à la fois sur LASeR et sur SVG.

Nous avons également implémenté les modules de composition et de rendu de scènes SVG conformément aux algorithmes présentés précédemment, notamment concernant la gestion des éléments temporels et des éléments graphiques. Nous avons validé la faisabilité d'un lecteur conforme à la norme SVG (avec les limitations présentées précédemment) qui allie la rapidité d'affichage et la faible utilisation mémoire. Nous avons également montré l'impact important sur toutes les parties d'un lecteur SVG dû à l'implémentation conjointe des notions d'héritage et d'animation.

Ce lecteur à l'heure actuelle est conforme à plus de 87% des séquences de conformité SVG Tiny 1.1, le reste étant lié au traitement des polices de caractères au format SVG, que nous n'avons pas étudié. Il a été validé sur PC, PDA et téléphone mobile comme le montre la Figure 7.6. De plus, même si aucune séquence de conformité SVG Tiny 1.2 n'existe au moment où ce document est rédigé, le lecteur

permet la lecture synchronisée, conformément à la norme SMIL, de contenu mixte SVG/audio/vidéo. Ces travaux nous ont conduit à soumettre de nombreux commentaires dans le cadre du processus de standardisation SVG notamment sur la consommation mémoire et les interfaces de programmation (voir section 9.5).

Figure 7.6 – Captures d'écran du lecteur SVG du projet GPAC

Enfin, cette implémentation nous a permis notamment de réaliser le parallèle entre les algorithmes de composition de scènes SVG et de scènes BIFS. Nous avons réalisé une implémentation qui, s'appuyant sur des moteurs de composition légèrement différents mais sur un moteur de rendu unique, permet la présentation de scènes SVG et BIFS. En effet, au niveau du module de composition, le traitement des éléments temporels du langage SVG est effectué au même moment que le traitement des nœuds `TimeSensor`. Le traitement des évènements peut également être effectué en parallèle dans le modèle VRML et DOM sans effet de bord et la gestion des média fonctionne de façon similaire. Ainsi, nous avons également pu avec succès valider la lecture de contenu multimédia animé interactif mixte BIFS et SVG.

Chapitre 8 Conclusion

8.1 Bilan

Au travers de ce document, nous avons défini, dans le chapitre 1, la notion de descriptions de scènes comme un moyen déclaratif de décrire l'utilisation des média, l'organisation spatiale, l'organisation temporelle et l'interactivité dans une présentation multimédia. Nous avons présenté les modèles de représentations existants, ainsi que les langages, standards officiels ou de fait, qui coexistent actuellement. Nous avons ensuite, dans le chapitre 2, présenté une classification des méthodes d'animations en deux catégories, l'animation par mise à jour et par interpolation, puis comparé les mérites de chaque méthode. Dans le chapitre 3, nous avons présenté les problèmes et les solutions existantes dans le domaine de la compression des descriptions de scènes, sous deux aspects : la compression de la structure et des données des scènes multimédia. Dans le chapitre 4, nous avons présenté les principes des mécanismes d'interactivité utilisés dans les scènes multimédia. Nous avons également mis en correspondance les modèles évènementiels DOM et VRML. Dans le chapitre 5, nous avons replacé les descriptions de scènes dans le contexte des chaînes de distribution de contenu multimédia en présentant les contraintes que les processus de création et de diffusion induisent.

Dans la seconde partie de ce document, nous avons décrit des propositions que nous avons été amenés à faire au cours de nos travaux. Nous avons apporté plusieurs optimisations aux mécanismes existants, que nous avons validées par des implémentations logiciels, que ce soit dans le domaine de la compression, notamment sur l'utilisation des mécanismes de quantification; de l'animation, pour la représentation de dessins animés vectoriels; de l'interactivité, pour l'utilisation de périphériques divers dans la norme MPEG-4; de l'efficacité de lecture, notamment pour les scènes SVG et dans une approche de type *streaming*. Il faut noter que ces travaux ont tous eu un impact positif fort sur la normalisation internationale puisqu'ils ont conduit à la définition de nouveaux outils et l'inclusion de ces nouveaux dans les standards SVG, BIFS ou LASeR.

Nous concluons ce document en détaillant les choix technologiques que nous ferions pour la conception d'un service multimédia pour mobile. Nous rappelons tout d'abord les contraintes que nous nous imposons.

8.2 Outils de descriptions de scènes et services multimédia pour mobiles

Comme nous l'avons indiqué, l'édition de contenu multimédia obéit à des contraintes spécifiques, notamment pour ce qui concerne l'édition de la description de la scène : utilisation de fichiers, souvent XML, en général non compressés, pour faciliter la compréhension d'un analyseur de contenu humain ou la réutilisation sous forme de gabarits; factorisation des propriétés communes entre scènes dans des feuilles de styles pour permettre l'édition efficace d'un ensemble de scènes; utilisation des mécanismes d'héritage de propriétés pour appliquer un style à une présentation. A l'opposé, la distribution de contenu multimédia pour des terminaux mobiles impose des contraintes différentes, parfois en opposition avec les contraintes précédentes : nécessité de réduction des débits, nécessité d'utiliser des mécanismes de *streaming* pour la construction de scènes incrémentales ou encore nécessité d'optimiser les traitements à effectuer au moment de la lecture.

Pour qu'un service de contenu multimédia pour mobile fonctionne efficacement, c'est-à-dire que la création de service soit simple et puissante, et que la lecture soit efficace, il nous semble important de satisfaire toutes ces contraintes. Cependant, si on continue l'analogie, abordée au chapitre 5, avec le monde de la vidéo numérique, l'édition de vidéo obéit à des contraintes qui sont également différentes de celles de sa diffusion. Le montage de vidéo numérique s'effectue plus facilement sur des flux non compressés mais il ne viendrait à l'idée de personne de diffuser de la vidéo numérique non compressée au format Haute Définition sur des réseaux de téléphonie mobile. Nous pensons donc qu'il est nécessaire, pour satisfaire les deux catégories de contraintes (édition et diffusion), de faire la distinction entre langage de descriptions de scènes multimédia pour l'édition et langages de descriptions de scènes multimédia pour la diffusion. Cela signifie notamment qu'une étape de publication doit être introduite pour passer d'un format d'édition à un format de diffusion.

8.2.1 Outils d'édition, outils pour la lecture et phase de publication

Nous récapitulons ici les outils présentés dans les chapitres précédents en précisant s'ils sont adaptés ou non à la phase d'édition et/ou de lecture; en indiquant le cas échéant, le rôle de la phase de publication et en présentant éventuellement les améliorations nécessaires à apporter aux technologies existantes pour y parvenir.

Diffusion de scènes multimédia

Domaine	Outils pour la lecture	Outils pour l'édition	Rôle de la phase de publication
Diffusion	Description sous forme de flux	Description sous forme de document	Fragmentation temporelle du document Création de point d'accès aléatoire Répartition de débit
	Compression permettant une lecture rapide et une diffusion en continue	Utilisation de langages XML	Détermination des paramètres de quantification par objet et des endroits optimaux pour leurs signalisations

Flux ou document

La sélection des outils de descriptions de scènes, pour la diffusion et la lecture, doit avoir pour objectif de faciliter la transmission sur les réseaux et de faciliter la lecture sur les terminaux cibles. Dans cette optique, nous pensons que la structure de la scène la plus appropriée est la structure sous forme de flux. En effet, même s'il est possible, comme nous l'avons montré, de diffuser en continu (en *streaming*) certains documents de scènes, l'utilisation d'un flux, composé, entre autres, de mises à jour d'insertion, de suppression, et de remplacement d'éléments de la scène, permet une diffusion plus souple (possibilité d'insertion d'objets ailleurs qu'à la fin du document, possibilité de retransmission en cas d'erreur). Parmi les outils existants, du fait qu'il est plus étendu que celui de la norme BIFS et que celui du langage Flash, nous retiendrions le langage de mise-à-jour de la norme LASeR pour concevoir un service multimédia pour mobiles à base de flux de descriptions de scènes.

Recommandation 1 : Utilisation de flux de description au format LASeR

Compression ou non

Nous pensons également que la compression, selon les algorithmes présentés dans les chapitres précédents, c'est-à-dire utilisant, d'une part, une compression sans perte et avec des tables de codage réduites pour le codage de structure, et d'autre part, une compression avec perte et à granularité de signalisation fine pour les données, est un outil indispensable à prendre en compte dans un service multimédia mobile car il permet de réduire le débit nécessaire sur les contenus volumineux, d'accélérer la lecture des scènes (volumineuses ou non), et enfin de réduire les latences d'accès pour les scènes transmises par les mécanismes de carrousel. Ainsi, nous proposerions d'utiliser une signalisation telle que définie par la norme MPEG-4 BIFS mais avec un algorithme de quantification, tel que nous l'avons décrit, à mi-chemin entre BIFS et LASeR, permettant une quantification efficace et un décodage sur des architectures à virgule fixe.

Recommandation 2 : Utilisation des mécanismes de codage avec et sans perte proposés dans le chapitre 3.

Recommandation 3 : Utilisation de la signalisation des paramètres de quantification selon la norme MPEG-4 BIFS.

Structuration

Domaine	Outils pour la lecture	Outils pour l'édition	Rôle de la phase de publication
Structuration	Arbre de scène simplifié (dictionnaire, interactivité, animation et sous-arbre d'affichage)	Arbre de scène non restreint	Analyse de la scène et simplification de la structure Gestion des objets du dictionnaire (suppression des objets inutiles) Séparation en sous-scènes
	Objets indivisibles, statiques et non modifiables	Objets non restreints	Analyse et signalisation des objets statiques

Structuration des objets

Dans le précédent chapitre, nous avons décrit les principales difficultés rencontrées pour la lecture, la composition et le rendu de la scène, c'est-à-dire la difficulté d'identifier dans l'arbre de scène les éléments qui sont modifiés entre deux cycles de présentation. Ces difficultés proviennent, entre autres, de la possibilité offerte par la manipulation potentiellement illimitée d'arbres de scène (notamment par script). N'importe quel élément peut être modifié à n'importe quel instant. La Figure 8.1 illustre les différentes représentations d'un objet de la scène nécessaires pour son affichage. Une représentation de l'objet (A) est lue à partir de la description de la scène, puis interprétée en une représentation native (B), et qui produit au final une représentation sous forme de pixels (C) pour le rendu visuel. Cependant, du fait des manipulations possibles sur la représentation A, (c'est-à-dire telle qu'elle est décrite dans le langage de description de la scène), il n'est pas possible, par exemple, de libérer l'espace mémoire occupé par la représentation A. Ensuite, dans certains langages, il n'est pas possible d'identifier les parties visuelles pour lesquelles le caractère vectoriel est inutile (car aucun zoom ne sera nécessaire), et donc de ce fait, il n'est pas possible de ne conserver que la représentation C de l'objet. Ceci nous conduit à donner la recommandation suivante.

Recommandation 4 : Utilisation d'un marquage des objets pour lequel le caractère vectoriel est superflu.

Enfin, la possibilité d'animer ou d'interagir avec n'importe quel objet dans la scène impose des contraintes sur les optimisations à réaliser : il faut par exemple respecter les parcours de capture et de bouillonnement dans le modèle DOM Events. Pour toutes ces raisons, nous préconisons l'utilisation d'une représentation de la scène qui permet d'indiquer explicitement que des parties de la scène, dites

immuables, ne sont ni accessibles par script, ni animables, ni modifiables par mise à jour, et sont atomiques vis-à-vis des mécanismes d'interactivité. Cela revient à considérer des parties de la scène comme des éléments du dictionnaire au sens du langage Flash.

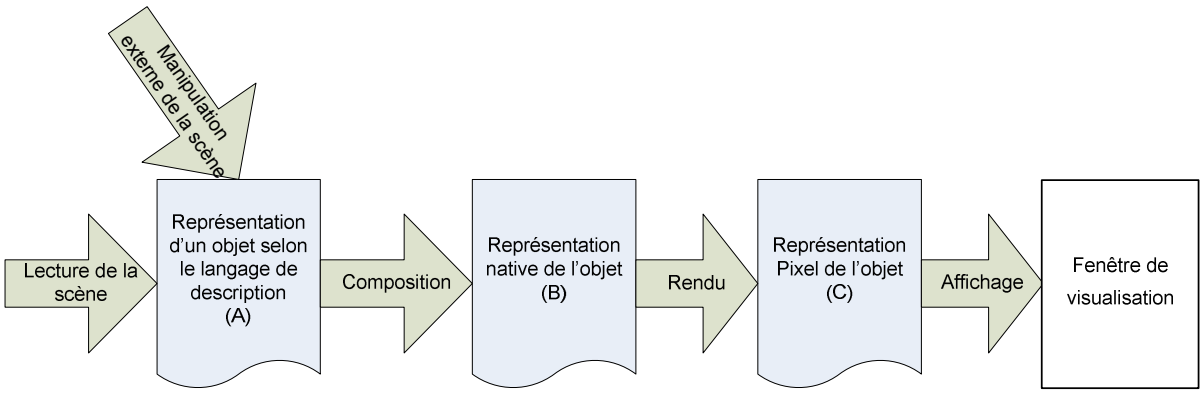


Figure 8.1 – Différentes représentation des objets d'une scène

Cette fonctionnalité permet au lecteur de scènes de construire la représentation nécessaire pour le rendu sans avoir ni à conserver, ni à parcourir des parties entières de la scène. La Figure 8.2 illustre une simplification possible d'un arbre de scène. Ainsi, dans une scène complexe, si certains nœuds de l'arbre (ici hachurés) indiquent que les sous arbres, dont ils sont les racines, sont immuables, c'est-à-dire qu'ils ne sont en aucun cas manipulables, alors le lecteur de scènes pourra ne conserver qu'une version optimisée, compacte, des sous arbres concernés : la représentation native (B) ou pixel (C), en fonction de l'utilisation du marqueur concernant le caractère vectoriel. Nous ajoutons donc la recommandation suivante.

Recommandation 5 : Utilisation d'un marquage explicite des objets immuables.

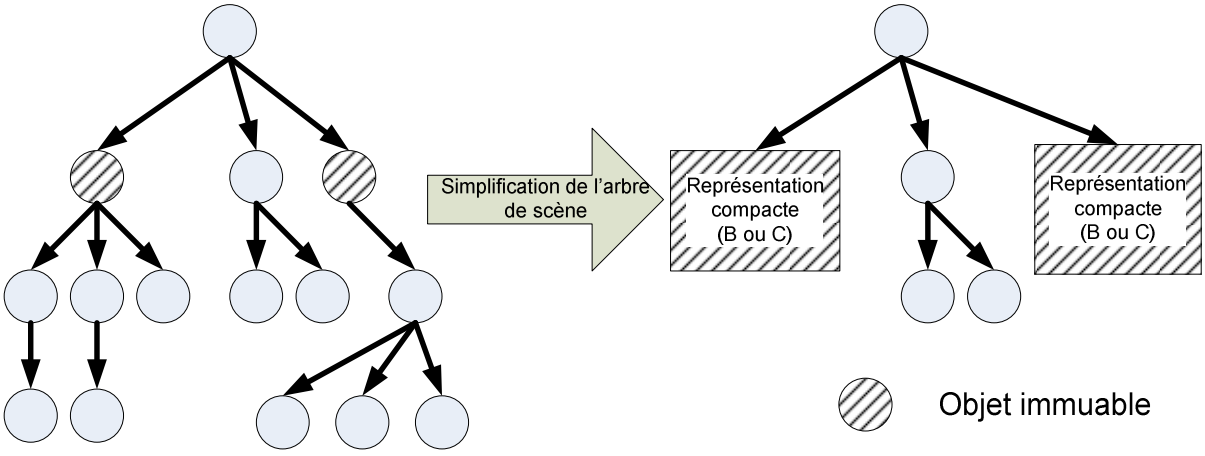


Figure 8.2 – Simplification d'un arbre de scène

Structuration de la scène

Concernant la structuration de la scène, la structuration sous forme de deux branches (dictionnaire et liste d'affichage), issue du langage Flash, présente l'avantage de la simplicité de traitement : à chaque cycle de présentation, seule la liste d'affichage est parcourue et l'identification des parties de la scène modifiée est immédiate. L'inconvénient de cette représentation est qu'elle n'est pas adaptée à l'utilisation de mécanismes déclaratifs pour l'interactivité ou l'animation. De plus, elle est limitée en termes d'efficacité de représentation : animer deux objets distincts nécessite d'animer deux couches de la liste d'affichage alors qu'une représentation arborescente permet de grouper les objets et d'utiliser une unique animation. Cette limitation est levée par l'introduction de *sprites* dans le langage Flash.

Sur la base de ces constatations, nous préconisons l'utilisation de scènes arborescentes et hiérarchiques, permettant l'utilisation de sous scènes, mais dont la structure est relativement rigide pour permettre une lecture efficace. Ainsi, nous proposons que chaque arbre de scène se décompose, de façon similaire à la composition des éléments *bindings* dans le langage XBL2, selon 4 branches : une branche contenant la définition des objets utilisés dans la scène (objets immuables ou non), cette branche, correspondant au dictionnaire, n'est jamais parcourue pendant le cycle de présentation (ni héritage, ni animation, et ne nécessite aucune traversée); une branche contenant les éléments d'interactivité; une branche contenant les éléments d'animation et autres éléments temporels; et enfin, une branche correspondant à une liste d'affichage étendue à la notion de sous-arbre d'affichage, notamment pour permettre une plus grande souplesse dans le positionnement spatial des objets. Le Code 8.1 décrit la structure d'une telle scène.

```
<Scene>
  <Dictionnaire>
    <Styles> … </Styles>
    <Objet id="O1" immuable="vrai">
      … description graphique …
    </Objet>
    <Objet id="O2" immuable="vrai" vectoriel="faux">
      … description graphique sans besoin de représentation vectorielle …
    </Objet>
    <Objet id="O3" immuable="faux">
      … sous arbre manipulable par script, animation, mise à jour …
      <… id="O3.1" …/>
      …
    </Objet>
    …
  </Dictionnaire>
  <Affichage>
    … primitives d'organisation spatiale …
  </Affichage>
  <Interactions>
   … écouteurs et gestionnaires d'évènements …
   … scripts référençant des objets non-immuables …
  </Interactions>
  <Timing>
   … primitives d'organisation temporelle …
  </Timing>
</Scene>
```

Code 8.1 – Structure de scène optimale pour la lecture

Recommandation 6 : Utilisation d'une structuration de la scène en quatre branches (dictionnaire, interactions, timing, affichage) selon le Code 8.1.

Comme nous l'avons indiqué au chapitre 3, l'utilisation de feuilles de style peut être intéressante pour factoriser les valeurs redondantes dans une scène et ainsi faciliter la compression. La norme CSS définit comment associer des styles à des éléments précis d'un document XML soit par le mécanisme de sélecteur soit par l'utilisation de l'attribut `class`. L'utilisation de sélecteur nécessite un traitement supplémentaire par rapport à l'utilisation de l'attribut `class`. Ce traitement consiste à appliquer la feuille de style à la scène : il s'agit de la phase de cascade. Cette étape nous paraît inutile pour satisfaire le besoin de factorisation des styles, c'est pourquoi nous recommandons l'utilisation de style à l'aide de l'attribut `class` dans les objets du dictionnaire, référençant un ensemble de style défini globalement au niveau du dictionnaire.

Recommandation 7 : Utilisation d'un élément `Styles` au niveau du dictionnaire regroupant les styles et référencé par un attribut `class` par les éléments du dictionnaire.

Organisation spatiale

Domaine	Outils pour la lecture	Outils pour l'édition	Rôle de la phase de publication
Organisation spatiale	Positionnement déterminé des objets Utilisation minimale de positionnement implicite	Organisation spatiale libre, préférablement par contrainte, par layout	Résolution du positionnement spatial des objets Détermination des zones de positionnement implicite

Considérant l'organisation spatiale des éléments de la scène, étant donné que le positionnement spatial des objets de la scène sous forme matricielle est un traitement simple à effectuer pour le lecteur, il est à privilégier par rapport au positionnement implicite (par contraintes). De plus, dans la plupart des cas, la phase de publication peut déterminer les transformations matricielles de chaque élément à partir du positionnement implicite global d'un ensemble d'éléments. Par contre, les mécanismes implicites de positionnement spatial (comme le *flow layout*) présentent un intérêt pour des situations bien particulières : positionnement et animation d'une séquence d'objets non déterminés à l'avance ou encore positionnement de texte en cours de saisie. Ces mécanismes ne doivent donc pas être exclus des descriptions de scènes de services multimédia mais ils doivent être utilisés dans une scène avec parcimonie. Ainsi, nous préconisons, pour le sous-arbre d'affichage, l'utilisation des éléments `g` du langage SVG (ou de façon équivalente, les éléments `Transform2D` ou `TransformMatrix2D` du langage BIFS) pour le positionnement explicite et l'utilisation d'éléments équivalents aux éléments `Layout` ou `Form` du langage BIFS pour le positionnement implicite, c'est-à-dire d'un positionnement par contraintes.

Recommandation 8 : Utilisation d'une organisation spatiale de la scène principalement matricielle avec possibilité d'utiliser des organisations implicites.

Organisation temporelle

Domaine	Outils pour la lecture	Outils pour l'édition	Rôle de la phase de publication
Organisation temporelle	Base de temps unique	Multiples bases de temps	Séparation de la scène en sous-scènes à base de temps unique
	Activation simplifiée des éléments temporels	Organisation temporelle souple	Création de *timers* et de routes entre objets temporels dépendants

Concernant l'organisation temporelle, la complexité du modèle de temps SMIL implique un nombre important de traitements lors d'un cycle de présentation. La possibilité de maintenir plusieurs bases de temps dans une même scène ne nous semble pas suffisante pour justifier la complexité du traitement d'un arbre avec plusieurs bases de temps. Ceci nous pousse à préférer un modèle plus simple reposant sur une seule base de temps par scène (MPEG-4 BIFS ou SVG). De plus, la possibilité de spécifier plusieurs temps d'activation/désactivation par élément temporel, y compris en indiquant des instants non résolus liés à des évènements, introduit une complexité inutile. Nous préconisions la suppression de la gestion des intervalles multiples et l'utilisation d'éléments spécifiques pour décrire les relations d'activation ou désactivation d'un élément temporel par un autre élément. Ce lien peut être matérialisé par un élément `listener` du langage SVG ou par une route dans le langage BIFS.

Recommandation 9 : Utilisation d'une unique base de temps par scène.

Recommandation 10 : Suppression de la gestion d'intervalles multiples d'activation/désactivation grâce à l'utilisation d'un élément explicite `listener` pour décrire les relations temporelles.

Animation

Domaine	Outils pour la lecture	Outils pour l'édition	Rôle de la phase de publication
Animation	Animations ciblées par l'utilisation de route	Animations de groupe	Création des routes issues d'animations de groupes

Dans la continuité de ce raisonnement, le modèle d'animation SMIL, surtout combiné à la notion d'héritage, nous semble également surdimensionné par rapport aux contraintes des descriptions de scènes lors de la phase de lecture. En effet, les notions de valeur spécifiée, valeur calculée et valeur de présentation induisent soit une complexité accrue des algorithmes de composition, soit une augmentation de la consommation mémoire. Ce modèle est complexe pour offrir principalement deux possibilités, certes intéressantes, à savoir : la composition d'animations et l'animation de propriétés de

groupe. La première est réalisable dans le modèle BIFS/VRML à l'aide d'un nœud équivalent au nœud `Valuator`, à savoir un nœud qui peut recevoir et combiner plusieurs évènements d'entrée en un évènement de sortie, avec un contrôle temporel. La seconde est également réalisable en multipliant les routes issues du nœud d'animation. Ces deux techniques présentent l'avantage de ne pas nécessiter le parcours de l'arbre pour propager le résultat d'une animation. Le langage de descriptions de scènes que nous préconisons se base, de ce fait, sur l'animation selon le modèle VRML plutôt que sur le modèle SMIL.

Recommandation 11 : Suppression du mécanisme d'héritage par l'introduction de la possibilité de cibler plusieurs éléments.

Recommandation 12 : Utilisation d'éléments spécifiques permettant la combinaison de plusieurs animations.

Interactivité

Domaine	Outils pour la lecture	Outils pour l'édition	Rôle de la phase de publication
Interactivité	Propagation simplifiée des évènements par route	Propagation des évènements au travers de l'arbre	Simulation des propagations ascendante et descendante
	Utilisation minimale de script et restrictions sur les interfaces d'accès		

L'interactivité, comme nous l'avons décrit, consiste à capturer et propager des évènements et à modifier la scène en conséquence. Les modèles DOM Events et VRML sont très proches, comme nous l'avons vu, pour la partie capture et propagation des évènements. A nouveau, nous estimons cependant que le modèle DOM Events, qui comprend les phases de capture, ciblage et bouillonnement, nécessite un traitement de la scène plus important que celui nécessaire au traitement de la scène équivalente dans le modèle VMRL. Ce dernier est préférable, à notre sens, car il définit explicitement, grâce aux routes, la relation entre émetteur et récepteur d'évènement et, de ce fait, ne nécessite pas le parcours de l'arbre de scène.

Recommandation 13 : Suppression des propagations ascendantes et descendantes implicites et utilisation d'éléments `listener` indiquant la propagation explicite des évènements.

Concernant les modifications de la scène suite à un évènement, les deux grandes approches que nous avons présentées sont l'utilisation de langages de script et l'utilisation de mises à jour de la scène. Il est évident que le mécanisme de mise à jour est à privilégier, car l'exécution de code de script est plus lent, plus coûteux en mémoire et nécessite de pouvoir naviguer dans la scène, ce qui implique que certaines optimisations dans la représentation de la scène ne sont pas possibles. Cependant, de nombreuses situations ne permettent pas de déporter les traitements complexes, qui ne peuvent être

effectués que par script, au niveau du serveur (diffusion en mode *broadcast* par exemple). Un langage de descriptions de scènes pour services multimédia doit pouvoir intégrer l'utilisation de script avec des restrictions sur les accès (lecture/écriture/navigation) dans l'arbre de scène.

Recommandation 14 : Utilisation de l'élément `Conditional` du langage LASeR pour réduire l'utilisation de code ECMAScript.

8.2.2 Limitations

Cette séparation de l'édition et de la distribution introduit quelques limitations. La première limitation concerne la réutilisation de contenu. En effet, la phase de publication introduit nécessairement une perte d'information (notamment les consignes de l'auteur). L'édition d'un contenu diffusé nécessitera une analyse du contenu pour retrouver les/des structures de scènes adaptées à l'édition. Cette analyse peut s'avérer difficile à effectuer à postériori. Cependant, on peut envisager, sur les réseaux possédant les débits suffisants, l'ajout de métadonnées permettant de faciliter cette analyse. Ces métadonnées peuvent par exemple être : la liste des animations qui étaient groupées, les scènes dont les propriétés de style étaient communes avant publication.

L'adaptabilité des contenus peut également souffrir de cette séparation. En effet, l'utilisation de technologies, réservée à notre sens à l'édition (comme la partie positionnement de la norme CSS) pour adapter le positionnement des objets en fonction du terminal cible, grâce à l'utilisation de *CSS Media Queries* par exemple, est de plus en plus répandu. La restriction de ces technologies à la phase d'édition empêche l'adaptation automatique au niveau du terminal client. Cette limitation peut être à notre sens réduite ou supprimée par l'utilisation de formats de descriptions de scènes scalables, au sens de la norme MPEG-21.

8.3 Perspectives

Durant les quelques années qui viennent de s'écouler, la situation dans le monde du multimédia a énormément évoluée. L'absence ou l'immaturité initiale des technologies a laissé place au foisonnement de standards. Nous avons eu la chance d'être témoins de cette évolution. L'effervescence se situe maintenant dans les instances de régulation, dans les organismes standardisant des applications complètes, dont le but est la sélection de la ou des technologies les plus appropriées aux nouveaux marchés que sont la téléphonie mobile et la télévision numérique terrestre. Le terme convergence souvent galvaudé prend tout son sens dans le domaine que nous avons abordé puisque les mêmes technologies de présentation peuvent être utilisées sur le Web, dans les décodeurs de télévision numérique ou sur les téléphones mobiles. Seules les problématiques de performance et d'usage différencient maintenant ces domaines.

Malgré cette convergence des technologies, la diversité subsiste dans les formats déployés. Il nous semble important que des travaux soient menés pour permettre le développement de passerelles de

transcodage automatique entre formats de descriptions de scènes, notamment pour traduire les mécanismes d'interactivité, sans nul doute, la partie la plus dure du transcodage de contenu multimédia d'un format à un autre.

Comme nous l'avons indiqué dans ce document, l'étude des performances des algorithmes actuels doit être continuée si l'on veut permettre réellement la sélection des technologies les plus appropriées à un contexte d'utilisation sur des bases techniques et non économiques. L'adéquation d'une technologie avec un terminal passe également par le développement de brique matérielles permettant l'allègement de la tâche des processeurs principaux des terminaux.

Enfin, face à la diversité des terminaux, des environnements de consommation, et au coût de création et de la publication de contenu multimédia, le développement de méthodes d'édition de contenu adaptable aux contextes d'utilisation (statique ou dynamique) est une piste de recherche également importante à suivre. Un travail intéressant consisterait à fournir des mécanismes d'adaptation permettant de changer l'interaction d'un contenu en fonction du terminal sur lequel il joue.

Chapitre 9 Liste des publications

9.1 Articles de revues

[01] G. Di Cagno, C. Concolato, J.-C. Dufourd, "Multimedia adaptation in end-user terminals", *Signal Processing: Image Communication, Volume 21, Issue 3, Mar. 2006, Pages 200 – 216.*
[02] J.-C. Dufourd, O. Avaro, C. Concolato, "An MPEG standard for rich media services", *IEEE Multimedia, Volume 12, Issue 4, Oct.-Dec. 2005, Pages 60 – 68.*
[03] C. Concolato, J.-C. Dufourd, J.-C. Moissinac, "Creating and encoding of cartoons using MPEG-4 BIFS: methods and results", *IEEE Transactions on Circuits and Systems for Video Technology, Volume 13, Issue 11, Nov. 2003, Pages 1129 – 1135.*

9.2 Articles de conférences

[04] M. Ransburg, H. Hellwagner, B. Pellan, C. Concolato, R. Cazoulat, S. De Zutter, C. Poppe, R. Van de Walle, A. Hutter, "DANAE: Dynamic and Distributed Adaptation of Scalable Multimedia Content in a Context-Aware Environment", *Proceedings of European Symposium on Mobile Media Delivery (EuMob), Alghero, Italy, Sept. 2006.*
[05] C. Concolato, J.-C. Dufourd, "Adaptation de contenu MPEG-4 BIFS suivant la norme MPEG-21", *Proceedings of Première Conférence sur le Multimédia pour Mobile (MCube), Montbéliard, France, Mar. 2004.*
[06] P. Gioia, K. Kamyab, I. Wolf, G. Panis, A. Difino, M. Kimiaei, T. DiGiacomo, A. Cotarmanac'h, P. Goulev, A. Graffunder, A. Hutter, B. Negro, C. Concolato, C. Joslin, E. Mamdani, J.-C. Dufourd, N. Thalmann, "ISIS: intelligent scalability for interoperable services", *Proceedings of First European Conference on Visual Media Production (CVMP), London, UK, Mar. 2004.*
[07] C. Concolato, J.-C. Moissinac, J.-C. Dufourd, "Representing 2D Cartoons using SVG", *Proceedings of SMIL Europe 2003, Paris, France, Feb. 2003.*
[08] C. Concolato, J.-C. Dufourd, "Comparison of MPEG-4 BIFS and some other multimedia description languages", *Proceedings of Workshop and Exhibition on MPEG-4 (WEMP), San Jose, USA, Jun. 2002.*
[09] J.-C. Moissinac, C. Concolato, J.-C. Dufourd, "Codage MPEG-4 de dessins animés", *Proceedings of COmpression et REprésentation des Signaux Audiovisuels (CORESA), Dijon, France, Nov. 2001.*
[10] F. Bouilhaguet, C. Concolato, S. Boughoufalah, J.-C. Dufourd, "Adding delivery support to MPEG-Pro, an authoring system for MPEG-4", *Proceedings of Workshop and Exhibition on MPEG-4 (WEMP), San Jose, USA, Jun. 2001.*

9.3 Rapports techniques et autres documents

[11] C. Concolato, "Specification of graphics for the mobile environment", *Delivrable 4.4.1, IST DANAE, Jun. 2006.*
[12] C. Concolato, "White Paper on MPEG-4 LASeR", *ISO/IEC JTC 1/SC 29/WG 11 N7507 Poznan, Poland, July 2005.*
[13] C. Concolato, "Generic scalability and streamability schemes specification", *Delivrable 3.7, IST ISIS, Jan. 2004.*

9.4 Brevets

[14] C. Concolato, J.-C. Dufourd, F. Prêteux, M. Preda, "Method and Equipment for Managing interaction in the MPEG-4 Standard".
[15] C. Concolato, C. Seyrat, G. Pau, C. Thiénot, A.Cotarmanac'h, "Method for compressing a hierarchical tree, corresponding signal and method for decoding a signal".

9.5 Contributions SVG et MPEG

C. Concolato, J. Le Feuvre, J.-C. Moissinac, “Feedback on SVG Traits”, http://lists.w3.org/Archives/Public/www-svg/2006Jan/0372.

C. Concolato, J. Le Feuvre, “SMIL Animation and Property Inheritance”, http://lists.w3.org/Archives/Public/www-svg/2006Sep/0003.

C. Concolato, J. Le Feuvre, “On the efficiency of display='none'”, http://lists.w3.org/Archives/Public/www-svg/2006Sep/0006.

C. Concolato, J. Le Feuvre, “On SAF global Streams”, *M13789, Hangzhou, Oct. 2006.*

C. Concolato, J. Le Feuvre, “On LASeR Waiting Tree”, *M13790, Hangzhou, Oct. 2006.*

J.-C. Dufourd, N. Pierre, E. Le Coq, C. Concolato, J. Le Feuvre, “Final word on the encoding of times in LASeR”, *M13969, Hangzhou, Oct. 2006.*

J. Le Feuvre, C. Concolato, “On LASeR fraction events”, *M13879, Hangzhou, Oct.2006.*

C. Concolato, “Text of 14496-20:DCOR (Editor's input)”, *M13729, Klagenfurt, July 2006.*

C. Concolato, J. Le Feuvre, “Proposed items for LASeR v2”, *M13244, Montreux, Mar. 2006.*

C. Concolato, J. Le Feuvre, “Items for Part 20 Corrigendum”, *M13243, Montreux, Mar. 2006.*

C. Concolato, J. Le Feuvre, “Request for clarification on Font Data Streams”, *M13242, Montreux, Mar. 2006.*

J. Le Feuvre, C. Concolato, “Conformance streams for MPEG-4 ATG”, *M13241, Montreux, Mar. 2006.*

C. Concolato, J. Le Feuvre, “Proposal for SAF and LASeR Conformance”, *M13235, Montreux, Mar. 2006.*

C. Concolato, J. Le Feuvre, “Technical analysis of the W3C SVG WG Liaison on LASeR”, *M13232, Montreux, Mar. 2006.*

J. Le Feuvre, C. Concolato, “LASeR FDIS Editorial Issues”, *M13018, Bangkok, Jan. 2006.*

C. Concolato, J. Le Feuvre, “Statistical analysis of LASeR conformance test and encoding comparisons”, *M12408, Poznan, Jul. 2005.*

C. Concolato, J. Le Feuvre, “Editorial and technical inputs for LASeR SoFCD”, *M12340, Poznan, Jul. 2005.*

Z. Lifshitz, C. Concolato, “Proposal for a free-distribution MAF”, *M12335, Poznan, Jul. 2005.*

C. Concolato, “Usage of LASeR content in MPEG-4 Systems environments”, *M12069, Busan, Apr. 2005.*

C. Concolato, P. de Cuetos, “An implementation of the MPEG-21 File Format”, M12067, *Busan, Apr. 2005.*

C. Concolato, P. de Cuetos, M. Kimiaei-Asadi, B. Pellan et al., “Report of CE on the use of AdaptationQoS for Conversions”, *M11884, Busan, Apr. 2005.*

P. de Cuetos, J.-C. Dufourd, C. Concolato, “Using BIFS updates for MELISA Sport Broadcasting System”, *M11671, Hong-Kong, Jan. 2005.*

C. Concolato, P. de Cuetos, M. Kimiaei-Asadi, B. Pellan, “Improvement of DIA Amd1: ConversionLink”, *M11670, Hong-Kong, Jan. 2005.*

C. Concolato, “Analysis of LASeR SoCD”, *M11375, Palma, Oct. 2004.*

C. Concolato, J. Le Feuvre, “Implementation of LASeR (and SVG) in GPAC”, *M11374, Palma, Oct. 2004.*

J.-C. Dufourd, O. Avaro, C. Concolato, “Support of SVG 1.2 in LASeR”, *M10684, Munich, Mar. 2004.*

J. Le Feuvre, C. Concolato, J.-C. Dufourd, “Various enhancements for MPEG-4 Systems”, *M10375, Hawai, Dec. 2003.*

J.-C. Dufourd, C. Concolato, J. Le Feuvre, “Requirements for a New Scene Description Format”, *M10110, Brisbane, Oct. 2003.*

J.-C. Dufourd, C. Concolato, J. Le Feuvre, “A Simpler, Efficient Binary encoding for MPEG-4 Scenes”, *M10091, Brisbane, Oct. 2003.*

J. Le Feuvre, C. Concolato, J.-C. Dufourd, “Various fixes on XMT-A schema”, *M10060, Brisbane, Oct. 2003.*

J. Le Feuvre, C. Concolato, J.-C. Dufourd, “Issues on externProto Coding”, *M10015, Brisbane, Oct. 2003.*

J. Le Feuvre, C. Concolato, J.-C. Dufourd, “Osmo4-GPAC description and demonstration”, *M09786, Trondheim, Jul. 2003.*

J. Le Feuvre, C. Concolato, J.-C. Dufourd, “Corrections to the BIFS ATG Extensions”, *M09753, Trondheim, Jul. 2003.*

S. Birman, Y. Elichai, E. Spiegel, C. Concolato, A. Cotarmanac’h, “Report on Synthesized texture coding and animation CE”, *M09750, Trondheim, Jul. 2003.*

S. Birman, E. Spiegel, C. Concolato, A. Cotarmanac’h, “Report on Synthesized texture CE”, *M09478, Pattaya, Mar. 2003.*

C. Concolato, J. Le Feuvre, “Comments on ATG extensions and implementation status”, *M09407, Pattaya, Mar. 2003.*

C. Concolato, J. Le Feuvre, J.-C. Dufourd, “Demonstration of Advanced Graphical BIFS” content, *M09248, Awaji Island, Dec. 2002.*

S. Birman, C. Concolato, A. Cotarmanac’h, “Report on Synthesized Texture CE”, *M09237, Awaji Island, Dec. 2002.*

J. Le Feuvre, C. Concolato, J.-C. Dufourd, “Improvements for WD2.0 of ATG”, *M09234, Awaji Island, Dec. 2002.*

C. Concolato, J. Le Feuvre, J.-C. Dufourd, “Request for a fix of the gradient nodes”, *M09156, Awaji Island, Dec. 2002.*

M. Preda, C. Concolato, F. Prêteux, “XML representation for Bone Based Animations”, *M09015, Shanghai, Oct. 2002.*

C. Concolato, J.-C. Dufourd, “Proposition of Test Set for the Core Experiment on Streaming Text”, *M08969, Shanghai, Oct. 2002.*

J. Le Feuvre, C. Concolato, J.-C. Dufourd, “MPEG-4 Systems Reference Software updates”, *M08932, Shanghai, Oct. 2002.*

C. Concolato, J.-C. Dufourd, J. Le Feuvre, “Improvements for WD 1.0 on ATG”, *M08929, Shanghai, Oct. 2002.*

C. Concolato, J.-C. Dufourd, “Systems demonstration more cartoons”, *M08619, Klagenfurt, Jul. 2002.*

C. Concolato, J.-C. Dufourd, Response to the CfP on Advanced Text and 2D Graphics”, *M08617, Klagenfurt, Jul. 2002.*

Y. Fisher, C. Concolato, O. Avaro, “AHG on MPEG-4 Systems 3rd Edition”, *M08493, Klagenfurt, Jul. 2002.*

C. Concolato, J.-C. Dufourd, “Use of negative values for Transform2D.scale”, *M08429, Fairfax, May 2002.*

C. Concolato, J.-C. Dufourd, “Preliminary response to the CfP on Advanced Text and 2D Graphics”, *M08353, Fairfax, May 2002.*

Y. Fisher, A. Cotarmanac’h, C. Concolato, “Report of the AHG on Advanced Text and Graphics”, *M08148, Jeju, Mar. 2002.*

C. Concolato, J.-C. Dufourd, “Comparison of the graphics tools of SVG and BIFS”, *M08029, Jeju, Mar. 2002.*

C. Concolato, J.-C. Dufourd, H. Grahn, “Proposition for new BIFS nodes”, *M7865, Pattaya, Nov. 2001.*

C. Concolato, J.-C. Dufourd, “Demonstration of ENST content production tools”, *M7850, Pattaya, Nov. 2001.*

C. Concolato, J.-C. Dufourd, “Changes to InputSensor and mapping of KeySensor and StringSensor”, *M7600, Pattaya, Nov. 2001.*

C. Concolato, “Report of the AHG on MPEG-4 BIFS”, *M7195, Sydney, Jul. 2001.*

C. Concolato, J.-C. Dufourd, Y. Fisher, “Multiple Documents Syndrome Fixes”, *M7189, Sydney, Jul. 2001.*

C. Concolato, J.-C. Dufourd, “Reference software implementation of the generic InputSensor node”, *M6949, Singapore, Mar. 2001.*

C. Concolato, J.-C. Dufourd, “Proposed amendment text for the generic InputSensor node”, *M6948, Singapore, Mar. 2001.*

J.-C. Dufourd, C. Concolato, F. Prêteux, M. Preda, “An extensible, generic, low-cost architecture for user interaction”, *M6811, Pisa, Jan. 2001.*

Chapitre 10 Bibliographie

10.1 Articles de revues, de conférences, thèses et livres

[16] A. Hakeem, K. Shafique, M. Shah, "An Object-based Video Coding Framework for Video Sequences Obtained From Static Cameras", *Proceedings of the 13th annual ACM international conference on Multimedia (MM), Singapore, Singapore, Nov. 2005.*

[17] A. Scherp, S. Boll, "Paving the Last Mile for Multi-Channel Multimedia Presentation Generation", *Procedings of the 11th Multimedia Modeling Conference (MMM), Melbourne, Australia, Jan. 2005*.

[18] E. Scheirer, "The MPEG-4 Structured Audio standard", *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Seattle, USA, May 1998.*

[19] W. De Neve, K. De Wolf, D. De Schrijver, R. Van de Walle, "Using MPEG-4 Scene Description for offering customizable and interactive multimedia presentations", *Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS), Montreux, Switzerland, Apr. 2005.*

[20] F. Bouilhaguet, J.-C. Dufourd, S. Boughoufalah, C. Havet, "Interactive broadcast digital television. The OpenTV platform versus the MPEG-4 standard framework", *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS), Geneva, Switzerland, May 2000.*

[21] J. Piesing, "The DVB multimedia home platform (MHP) and related specifications", *Proceedings of the IEEE, Volume 94, Issue 1, Jan. 2006, Pages 237 – 247.*

[22] R. Grigoras, "Supervision de flux pour les contenus hypermédia: optimisation de politiques de préchargement et ordonnancement causal", *Thèse de doctorat, INP Toulouse, 2003*.

[23] P. Baudelaire, M. Gangnet, "Planar maps: an interaction paradigm for graphic design", *Proceedings of the conference on Human factors in computing systems (CHI), Austin, USA, May 1989.*

[24] E. Jang et al., "Interpolator Data Compression for MPEG-4 Animation", *IEEE Transactions on Circuits And Systems for Video Technology, Volume 14, Issue 7, Jul. 2004, Pages 989 – 1008.*

[25] P.-J. Laurent, A. Le Méhauté, L. Schumaker, "Curves and Surfaces", *Academic Press, New York, 1991*.

[26] O. Avaro, P. Salembier, "MPEG-7 Systems: overview", *IEEE Transactions on Circuits and Systems for Video Technology, Volume 11, Issue 6, Jun. 2001, Pages 760 – 764.*

[27] S. Devillers, C. Timmerer, J. Heuer, H. Hellwagner, "Bitstream syntax description-based adaptation in streaming and constrained environments", *IEEE Transactions on Multimedia, Volume 7, Issue 3, Jun. 2005, Pages 463 – 470.*

[28] T. Ebrahimi, F. Pereira, "The MPEG-4 Book", *Prentice Hall, Jul. 2002.*

[29] C. Seyrat, "L'interopérabilité dans les systèmes d'indexation multimédia", *Thèse de doctorat, LIP6, 2003.*

[30] J. Ziv, A. Lempel, "A Universal Algorithm for Sequential Data Compression", *IEEE Transactions on Information Theory, Volume 23, Issue 3, 1977, Pages 337 – 342.*

[31] S. Boughoufalah, "Outils Auteurs pour MPEG-4", *Thèse de doctorat, ENST, 2002.*

[32] G. Di Cagno, "Systèmes multimédia et qualité d'expérience", *Thèse de doctorat, ENST, 2004*.

[33] M. Kostoulas, M. Matsa, N. Mendelsohn, E. Perkins, A. Heifets, M. Mercaldi, "XML screamer: an integrated approach to high performance XML parsing, validation and deserialization", *Proceedings of the 15th International Conference on World Wide Web (WWW), Edinburgh, Scotland, May 2006.*

[34] C. Cunat, "Accélération matérielle pour le rendu de scènes multimédia vidéo et 3D", *Thèse de doctorat, ENST, 2004*.

[35] Y.-C. Tu, J. Sun, M. Hefeeda, S. Prabhakar, "An analytical study of peer-to-peer media streaming systems", *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP), Volume 1, Issue 4, Nov. 2005, Pages 354 – 376.*

[36] A. Vetro, C. Timmerer, "Digital item adaptation: overview of standardization and research activities", *IEEE Transactions on Multimedia, Volume 7, Issue 3, Jun. 2005, Pages 418 – 426.*

[37] T. Zgaljic, N. Sprljan, E. Izquierdo, "Bitstream syntax description based adaptation of scalable video", *Proceedings of The 2nd European Workshop on the Integration of Knowledge, Semantics and Digital Media Technology (EWIMT), London, UK, Nov.-Dec. 2005.*

10.2 Articles divers

[38] Y.-K. Lim, "White Paper on MPEG-4 System", *ISO/IEC JTC 1/SC 29/WG 11N7504, 2005,* http://www.chiariglione.org/mpeg/technologies/mp04-sys/index.htm

[39] A. Cotarmanac'h, R. Cazoulat, Y. Fisher, " White Paper on MPEG-4 BIFS ", *ISO/IEC JTC 1/SC 29/WG 11N7608, 2005*, http://www.chiariglione.org/mpeg/technologies/mp04-bifs/index.htm

[40] P. Schirling, "White Paper on MPEG-2 Systems", http://www.chiariglione.org/mpeg/technologies/mp02-ts/index.htm

[41] J. Bormans, K. Hill, "MPEG-21 Overview", *ISO/IEC JTC1/SC29/WG11/N5231, 2002*, http://www.chiariglione.org/mpeg/standards/mpeg-21/mpeg-21.htm

10.3 Standards du Moving Picture Experts Group (MPEG)

[42] Information technology – Coding of audio-visual objects – Part 1: Systems, ISO/IEC 14496-1:2004

[43] Information technology – Coding of audio-visual objects – Part 11: Scene description (BIFS) and application engine (MPEG-J), ISO/IEC 14496-11:2005

[44] Information technology – Coding of audio-visual objects – Part 12: ISO base media file format, ISO/IEC 14496-12:2005

[45] Information technology – Coding of audio-visual objects – Part 14: MPEG-4 File Format (MP4), ISO/IEC 14496-14:2003

[46] Information technology – Coding of audio-visual objects – Part 20: Lightweight Application Scene Representation (LASeR) and Simple Aggregation Format (SAF), ISO/IEC 14496-20:2005

[47] Information technology – MPEG systems technologies – Part 1: Binary MPEG format for XML (BiM), ISO/IEC 23001-1:2006

10.4 Standards du World Wide Web Consortium (W3C)

[48] XHTML 1.0, The Extensible HyperText Markup Language (Second Edition), W3C Recommendation, 26 January 2000, revised 1 August 2002, http://www.w3.org/TR/html/

[49] Synchronized Multimedia Integration Language Specification (SMIL 2.1), W3C Recommendation, 13 December 2005, http://www.w3.org/TR/SMIL/

[50] Scalable Vector Graphics (SVG) Tiny 1.2 Specification, W3C Candidate Recommendation, 10 August 2006, http://www.w3.org/TR/SVGMobile12/

[51] Cascading Style Sheets, level 2 (CSS2) Specification, W3C Recommendation, 12 May 1998, http://www.w3.org/TR/REC-CSS2/

[52] Document Object Model (DOM) Level 3 Core Specification, W3C Recommendation, 07 April 2004, http://www.w3.org/TR/DOM-Level-3-Core/

[53] XForms 1.0 (Second Edition), W3C Recommendation, 14 March 2006, http://www.w3.org/TR/xforms/

[54] Extensible Markup Language (XML) 1.0 (Fourth Edition), W3C Recommendation, 16 August 2006, http://www.w3.org/TR/xml/

[55] SVG MicroDOM Specification, http://www.w3.org/TR/SVGMobile12/svgudom.html

[56] Compound Document Formats, http://www.w3.org/2004/CDF/

[57] XML Path Language (XPath), Version 1.0, http://www.w3.org/TR/xpath

[58] XML Pointer Language (XPointer), Working Draft, 16 August 2002, http://www.w3.org/TR/xptr/

[59] Remote Events for XML (REX) 1.0, Working Draft 02 February 2006, http://www.w3.org/TR/rex/

[60] Document Object Model (DOM) Level 3 Events Specification, Version 1.0, W3C Working Draft, 13 April 2006, http://www.w3.org/TR/DOM-Level-3-Events/

[61] Efficient XML Interchange Measurements Note, W3C Working Draft, 18 July 2006, http://www.w3.org/TR/exi-measurements/

[62] Analysis of the EXI Measurements,W3C Public Working Group Document, 20 July 2006, http://www.w3.org/XML/EXI/report.html

[63] XML Schema Part 0: Primer Second Edition, W3C Recommendation, 28 October 2004, http://www.w3.org/TR/xmlschema-0/

[64] XSL Transformations (XSLT), Version 1.0, W3C Recommendation, 16 November 1999, http://www.w3.org/TR/xslt

[65] XML Binding Language (XBL) 2.0, W3C Working Draft, 19 June 2006, http://www.w3.org/TR/xbl/

[66] SVG's XML Binding Language (sXBL), W3C Working Draft, 15 August 2005, http://www.w3.org/TR/sXBL

[67] XML Events, An Events Syntax for XML, W3C Recommendation, 14 October 2003, http://www.w3.org/TR/xml-events/

[68] The XMLHttpRequest Object, W3C Working Draft, 19 June 2006, http://www.w3.org/TR/XMLHttpRequest/

[69] Scalable Vector Graphics (SVG) 1.2, W3C Working Draft, 27 October 2004, http://www.w3.org/TR/2004/WD-SVG12-20041027/

10.5 Standards divers

[70] ECMAScript Language Specification, 3rd edition (December 1999), www.ecma-international.org/publications/standards/Ecma-262.htm

[71] The Virtual Reality Modeling Language (VRML) specification, http://www.web3d.org/x3d/specifications/vrml/ISO-IEC-14772-VRML97/

[72] The Extensible 3D (X3D) specification, http://www.web3d.org/x3d/specifications/ISO-IEC-19775-X3DAbstractSpecification/

[73] 3rd Generation Partnership Project, Technical Specification Group Services and System Aspects Transparent end-to-end packet switched streaming service (PSS); 3GPP file format (3GP) (Release 7), 3GPP TS 26.244, http://www.3gpp.org/ftp/Specs/archive/26_series/26.244/

[74] Handley M., "Session Description Protocol", RFC 2327, http://www.ietf.org/rfc/rfc2327.txt

[75] Schulzrinne H. et al., "RTP: A Transport Protocol for Real-Time Applications", http://www.ietf.org/rfc/rfc1889.txt

[76] T. Paila et al., "File Delivery over Unidirectional Transport (FLUTE)", http://www.ietf.org/rfc/rfc3926.txt

[77] J. van der Meer et al., "RTP Payload Format for Transport of MPEG-4 Elementary Streams", http://ietf.org/rfc/rfc3640.txt

[78] Streaming Transformations for XML, Site Officiel, http://stx.sourceforge.net/

[79] XBL - Extensible Binding Language 1.0, http://www.mozilla.org/projects/xbl/xbl.html

10.6 Logiciels et sites Internet

[80] Digital Media Broadcasting, http://en.wikipedia.org/wiki/Digital_Multimedia_Broadcasting

[81] Projet DANAE, Site Officiel, http://danae.rd.francetelecom.com/

[82] Projet ISIS, Site Officiel, http://isis.rd.francetelecom.com/

[83] "Asynchronous JavaScript And XML", http://en.wikipedia.org/wiki/Ajax_%28programming%29

[84] Document Object Model, http://fr.wikipedia.org/wiki/Document_Object_Model

[85] Flash, http://en.wikipedia.org/wiki/Adobe_Flash

[86] Dynamic HTML, http://en.wikipedia.org/wiki/Dynamic_HTML

[87] OGG, http://fr.wikipedia.org/wiki/Ogg

[88] OpenTV, Site Officiel, http://www.opentv.com

[89] Multimedia Home Platform, Site Officiel, http://www.mhp.org/

[90] Flash, Site Officiel, http://www.adobe.com/products/flash

[91] FlashLite, Site Officiel, http://www.adobe.com/products/flashlite/

[92] Toon Boom Studio, Site Officiel, http://www.toonboom.com/

[93] Adobe Illustrator, Site Officiel, http://www.adobe.com/fr/products/illustrator/

[94] Beatware Mobile Designer, Site Officiel, http://www.beatware.com/products/md.html

[95] FlashJax, http://www.flash-widgets.com/flash_jax.html

[96] GPAC Project on Advanced Content, http://gpac.sourceforge.net/

[97] 3D Studio Max, Site Officiel, http://www.autodesk.com

[98] Adobe Premiere, Site Officiel, http://www.adobe.com/fr/products/premiere/

[99] LibXML, Site Officiel, http://xmlsoft.org/

[100] Matroska, Site Officiel, http://www.matroska.org/

[101] ID3, Site Officiel, http://www.id3.org/

[102] QuickTime, Site Officiel, http://www.apple.com/fr/quicktime

[103] Darwin Streaming Server, Site Officiel, http://developer.apple.com/opensource/server/streaming/index.html

[104] ToonBoom, Site Officiel, http://www.toonboom.com

[105] Hypertext Transfer Protocol (HTTP), http://fr.wikipedia.org/wiki/Http

[106] Multimedia Messaging System (MMS), http://en.wikipedia.org/wiki/Multimedia_Messaging_Service

Chapitre 11 Glossaire

3GP	Format de fichier issu de la norme MPEG-4 et précisé par le consortium 3GPP pour le stockage de données multimédia pour la téléphonie mobile
3GPP	"Third Generation Partnership Program", consortium d'industriels définissant des normes pour la téléphonie mobile
3GPP Timed Text	Format de description de texte pour le sous-titrage défini par le consortium 3GPP
ActionScript	Ensemble d'interfaces pour la manipulation programmatique de contenu Flash
AJAX	"Asynchronous JavaScript And XML", ensemble de technologies permettant de créer des services interactifs plus réactifs pour Internet que les services traditionnels
AU	Unité d'accès minimale d'un flux de données à laquelle on peut associer un temps
AVI	Format de fichier usuel pour le stockage de flux audiovisuels
BIFS	"BInary Format for Scenes", premier format issu de la norme MPEG-4 pour le codage de descriptions de scènes, basé sur le langage VRML
BIFS-Update	Commande de modification d'une scène MPEG-4 encodée au format BIFS
BiM	Format de codage binaire de document XML standardisé dans la norme MPEG-7
BSD, gBSD	"(generic) Bitstream Description Language", standards de la norme MPEG-21 permettant la description au format XML de la structure d'un flux binaire
CDF	"Compound Document Format", norme définie par le W3C décrivant les interactions entre les langages XML pour la présentation de contenu Web
CSS	"Cascading Style Sheet", standard W3C pour la description du style, du positionnement d'une partie de l'interactivité des pages HTML
CSS Media Queries	Standard du W3C permettant l'adaptation d'un contenu par la sélection d'une feuille de style CSS en fonction des caractéristiques du client
DANAE	"Dynamic and distributed Adaptation of scalable multimedia coNtent in a

	context-Aware Environment", projet de recherche sur la creation d'une chaîne de diffusion et d'adaptation dynamique et distribuée de contenu multimédia selon la norme MPEG-21
DHTML	"Dynamic HTML", combinaison de contenu HTML et de programme Javascript pour l'animation de pages HTML
DOM	"Document Object Model", modèle et interface de manipulation des documents XML défini par le W3C
DOM 3 Events	Standard W3C (version 3) qui définit un modèle évènementiel associé aux documents XML
DSM-CC Data Carrousel	Technique définie par la norme MPEG-2 pour le transfert périodique de données
DVB	"Digital Video Broadcasting", ensemble de standards pour la télévision numérique européenne
DVB-H	"DVB Handheld", ensembe de standards pour la télévision numérique mobile
DVB-T	"DVB Terrestrial", ensemble de standards pour la télévision numérique terrestre, hertzienne
ECMAScript	Langage de programmation à la base de nombreux langages utilisés sur internet
EXI	Groupe de travail au sein du W3C sur la compression de documents XML
FLUTE	Protocole de diffusion de fichiers sur les réseaux internet, en mode *multicast*
FTP	"File Transfer Protocol", protocole historique de transfert de fichier sur internet
GPAC	"GPAC Project on Advanced Content", plateforme logicielle multimédia de l'ENST
GZIP	Outil de compression s'appuyant sur l'algorithme ZLIB
HTML	"HyperText Markup Language", langage informatique pour décrire les pages Internet
HTTP	"HyperText Transport Protocol", protocole de téléchargement de fichiers

	pour les réseaux internet
ID3	Standard de facto pour la description de métadonnées associées aux fichiers MP3
ISIS	"Intelligent Scalability for Interactive Services", projet de recherche sur l'adaptation de contenu multimédia interactif au contexte d'utilisation selon la norme MPEG-21
ISO	"International Organization for Standardization", institut de standardisation international
JPEG	"Joint Picture Experts Group", comité de standardisation de l'ISO dans le domaine du codage d'image
JXTA	Plateforme et protocole de transfert de fichiers utilisé dans les logiciels Peer-to-Peer
LASeR	"Lightweight Application Scene Representation", second format issu de la norme MPEG-4 pour le codage de descriptions de scènes, basé sur le langage SVG
Matroska	Format de fichier pour le stockage de flux audiovisuels défini par la communauté du logiciel libre
MELISA	"Multi-platform E-publishing for Leisure and Interactive Sports Advertising", projet de recherche sur la création de services multimédia et interactifs de paris sportifs
MHP	"Multimedia Home Platform", standard issu du groupe DVB pour la manipulation de scènes multimédia sur décodeur de télévision numérique
MicroDOM	Modèle et interface pour la manipulation simplifiée de document SVG
MIDI	Format audio permettant la description de flux musicaux synthétiques
MMS	"Multimedia Messaging System", protocole de téléchargement de fichiers pour les réseaux de téléphonie mobile
MOV	Format de fichier pour le stockage de flux audiovisuels utilisé par le logiciel QuickTime de la société Apple
MP3	Format de codage et format de fichier de flux musicaux conformes à la partie 3 de la norme MPEG-1
MP4	Format de fichier défini par la norme MPEG-4 pour le stockage de

	données multimédia au format MPEG-4
MP4MC	"Mobile Platform for Musical Content", projet de recherche sur la distribution de contenu musical protégé à destination des téléphones mobiles
MPEG	"Moving Picture Experts Group", comité de standardisation de l'ISO dans le domaine du codage audio, vidéo et des descriptions associées
MPEG-2	Norme produite par le groupe MPEG, comportant un format de codage vidéo, un format de codage audio et des formats de multiplexage
MPEG-2 TS	"Transport Stream", format de multiplexage de la norme MPEG-2 pour le transport de données audiovisuelles pour la télévision numérique
MPEG-21	Norme produite par le groupe MPEG, permettant la description des caractéristiques des flux audiovisuels nécessaires à la protection et à l'adaptation automatisée
MPEG-21 DIA	"Digital Item Adaptation", Partie de la norme MPEG-21 traitant de l'adaptation automatique de contenu multimédia
MPEG-4	Norme produite par le groupe MPEG, comportant entre autre deux formats de codage vidéo, un format de codage audio, deux formats de codage de descriptions de scènes et un format de fichier
MPEG-4 Systems	Ensemble des parties de la norme MPEG-4 qui traitent de la synchronisation, de l'interactivité, et du transport des flux audiovisuels
MPEG-7	Norme produite par le groupe MPEG, permettant la description des caractéristiques des flux audiovisuels nécessaires à l'indexation et à l'archivage
MPG	Format de fichier pour le stockage de flux audiovisuels au format MPEG-1 ou MPEG-2
multicast	Protocole permettant la diffusion simultanée et efficace sur Internet d'un même contenu à de nombreux utilisateurs
OGG	Format de fichier pour le stockage de flux audiovisuels défini par la communauté du logiciel libre
OpenTV	Solution propriétaire pour la manipulation de scènes multimédia sur décodeur de télévision numérique et la création de programmes interactifs
PSNR	"Peak Signal to Noise Ratio", outil de mesure de la qualité d'une image

	après traitement
RAP	"Random Access Point", Point d'accès aléatoire dans un flux ne nécessitant aucune autre information pour être décodé
REX	"Remote Events for XML", langage XML permettant de décrire des modifications à appliquer à un document XML
RIAM, PRIAMM	"Réseau d'Innovation en Audiovisuel et Multimédia", organisation française responsable de la labellisation de projet de recherche
RTP	"Real-time Transport Protocol", protocole utilisé pour la diffusion de flux audiovisuels sur les réseaux Internet
RTSP	"Real-Time Streaming Protocol", protocole utilisé pour le contrôle des sessions de *streaming* de flux audiovisuels sur internet
SAMP4	"Secure Audio for MPEG-4", projet de recherche sur la sécurisation de contenu audio au format MPEG-4
SAOL	"Structured Audio Orchestra Language", format audio, défini par la norme MPEG-4, permettant la description de flux de musique synthétique
SAX	"Simple API for XML", interface pour la lecture et l'écriture de document XML nécessitant une faible utilisation mémoire
Schema XML	Standard du W3C permettant la description en XML de la grammaire permettant de vérifier les règles de construction de documents XML
SDP	"Session Description Protocol", protocole utilisé pour la description d'une session de *streaming*
SMIL	"Synchronized Multimedia Integration Language", standard du W3C pour la description de contenu multimédia animé
SMIL Animation	Partie du standard SMIL définissant le modèle et les outils d'animation
SMIL Timing	Partie du standard SMIL définissant le modèle de temps et les outils de synchronisation
SMTP	"Simple Mail Tranfer Protocol", protocole d'échange des courriers électroniques
SoNG	"Portals of Next Generation", projet de recherche sur la création de portails de navigation multimédia réalisés en MPEG-4
SRT	Format de description de sous-titres utilisé par la communauté du logiciel

	libre
STX	"Streaming Transformation for XML", langage permettant de décrire des transformations de documents XML pouvant être effectuées au fur et à mesure de la lecture du document à transformer
SVG	"Scalable Vector Graphics", langage XML pour la description de scènes graphiques vectorielles animées défini par le W3C
SVG 1.2 Tiny	Restriction de la version 1.2 de la norme SVG retenue pour la téléphonie mobile
SWF	Format de fichier pour la publication de contenu multimédia produit par l'outil Flash de la société Macromédia/Adobe
sXBL	"SVG's XML Binding Language", tentative de standard W3C pour la définition d'un langage permettant l'association de primitives SVG pour la présentation de documents XML
T-DMB	"Terrestrial - Digital Media Broadcasting", protocole de distribution de contenu multimédia pour la télévision numérique Coréenne utilisant la norme MPEG-4 BIFS
TIRAMISU	"The Innovative Rights and Access Management Inter-platform Solution", projet de recherche sur la création d'une chaîne d'édition, diffusion et consommation de contenu multimédia protégé
VRML	"Virtual Reality Modelling Language", langage de description de mondes virtuels en trois dimensions
W3C	"World Wide Web Consortium", consortium d'industriels et d'universitaire pour la standardisation de technologies utiles à l'internet comme le langage HTML
X3D	Evolution du langage VRML pour la description de mondes virtuels en trois dimensions, s'appuyant sur une description XML
XBL	"XML Binding Language", langage d'abstraction pour associer des paramètres de présentation à des documents XML
XForm	Standard du W3C permettant l'intégration de formulaires dans les pages HTML
XHTML	Evolution du langage HTML s'appuyant sur le langage XML
XML Events	Standard du W3C permettant la description en XML de primitive pour

	l'écoute et le traitement d'évènements conforme à la forme DOM Events
XMLHttpRequest	Standard du W3C permettant aux créateurs de pages internet d'indiquer à un navigateur d'effectuer des requêtes automatiques à des serveurs de données XML
XMT, XMT-A, XMT-O	"eXtended MPEG-4 Textual format", ensemble de deux langages XML (XMT-O et XMT-A) pour la création de contenu MPEG-4
XPath	Standard du W3C pour indiquer un ou plusieurs chemins dans un arbre DOM
XPointer	Standard du W3C pour l'adressage d'une partie d'un document XML
XSL	"XML Style Sheet", standard du W3C permettant d'associer des styles aux documents XML
XSLT	"XML Style Sheet Transformation", standard du W3C permettant de décrire des transformations applicables aux documents XML
ZLIB	Algorithme générique de compression sans perte basé sur les travaux de Ziv et Lempel