A framework for distributed 3D graphics applications based on compression and streaming

Ivica Arsov

Thèse de doctorat de Télécom & Management SudParis dans le cadre de l'école doctorale S&I en co-accréditation avec
l' Universite d'Evry-Val d'Essonne

Spécialité :
Informatique

Par
M. Ivica ARSOV

Thèse présentée pour l'obtention du diplôme de Docteur
de Télécom & Management SudParis

A framework for distributed 3D graphics applications based on compression and streaming

Soutenue le 31 Mars 2011 devant le jury composé de :

Rapporteurs :
M. Jacques Fayolle, Maître de conférences, HDR, TELECOM Saint Etienne, France
M. Euee S. Jang, Professeur à l'Université Hanyang de Seoul, République de Corée

Examinateurs :
M. Vasile Buzuloiu, Professeur à l'Université Politehnica de Bucarest, Roumanie
M. Mohamed Daoudi, Professeur à Telecom Lille 1, France
M. Francisco Morán Burgos, Maître de conférences à l'Université polytechnique de Madrid, Espagne
M. Christian Timmerer, Maître de conférences à l'Université de Klagenfurt, Autriche

M. Bruno DEFUDE, Professeur à Telecom & Management SudParis, Directeur de thèse
M. Marius Preda, Maître de conférences, Telecom & Management SudParis, Co-encadrant

Thèse Numéro : 2011TELE0013

Acknowledgements

I would like to thank Prof. Françoise Préteux that lead my studies from 2006-2010.

I also owe my thanks to Prof. Bruno Defude, who accepted me as PhD student in the last year of the studies. I owe my utmost gratitude to Dr. Marius Preda who has supported me thoughout my thesis with his patience and knowledge whilst allowing me the room to work in my own way. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisors, I would like to thank the rest of my thesis committee: Prof. Dr. Jacques Fayolle, Prof. Dr. Euee S. Jang, Prof. Dr. Vasile Buzuloiu, Prof. Dr. Mohamed Daoudi, Dr. Francisco Morán Burgos and Dr. Christian Timmerer, for their encouragement, insightful comments, and hard questions.

It is a pleasure to thank all the people I worked with during the past years and made this thesis possible. Particularly, I would like to show my gratitude to my colleague and girlfriend, Dr. Blagica Jovanova, who has made wide available her support not only in the professional field, but also in the private life. Honestly, I thank her for putting up with me during the most difficult periods and to encourage me, and for all the emotional support, camaraderie, entertainment, and caring she provided.

I also want to thank my fellow lab mates in the ARTEMIS department, for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last years.

Last but not the least, I whould like to thank my familiy and for their continuous support in every aspect.

I dedicate this work to my grandfather, Ignat Ilievski, who passed away in 2010 and did not live to see my defense. He was there for me since my birth and always telling me to go forward and to improve myself. I am taking his advice.

Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Description
1G	First Generation of mobile networks
2D	Two Dimensions
2G	Second Generation of mobile networks
3D	Three Dimensions
3DMC	3D Mesh Coding
3G	Third Generation of mobile networks
AFX	Animation Framework Extension
AI	Artificial Intelligence
API	Application Programming Interface
AU	Access Unit
BBA	Bone Based Animation
BIFS	Binary Format for Scene
CAD	Computer-Aided Design
COLLADA	COLLAborative Design Activity
DAI	DMIF Application Interface
DCC	Digital Content Creation
DMIF	Delivery Multimedia Integration Framework
DVD	Digital Video Disc
EDGE	Enhansed Data rates for GSM Evolution
EDSAC	Electronic Delay Storage Automatic Calculator
ENIAC	Electronics Numerical Integrator and Computer
ES	Elementary Stream
ESD	Elementary Stream Descriptor
FAMC	Frame-Based Animated Mesh Compression
FFD	Free-Form Deformation
FPS	Frames Per Second
GPRS	General Packet Radio Service
GPS	Global Positioning System
GPU	Graphics Processing Unit
GSM	Groupe Special Mobile
GUI	Graphical User Interface
HSCSD	High-Speed Circuit-Switched Data
HSPA	High-Speed Downlink Packet Access

Abbreviation	Description
HSUPA	High-Speed Uplink Packet Access
IC	Integrated Circuits
IOD	Initial Object Descriptor
IPMP	Intelectual Property Management and Protection
ISO	International Standards Organization
JCL	Job Control Language
LASeR	Lightweight Application Scene Representation
LOD	Level Of Detail
M3G	Mobile 3D Graphics
MPEG	Moving Pictures Expert Group
MTA	Mobile System A
MTU	Maximum Transfer Unit
MXM	MPEG Extensible Middleware
OCI	Object Content Information
OD	Object Descriptor
ODF	Object Descriptor Framework
OS	Operating System
PC	Perosnal Computer
PDA	Personal Digital Assistant
RIA	Rich Internet Applications
RTP	Real-time Transport Protocol
RTSP	Real Time Streaming Protocol
SDM	System Decoder Model
SDP	Session Description Protocol
SMIL	Synchronized Multimedia Integration Language
SVG	Scalable Vector Graphics
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UMTS	Universal Mobile Telecommunication System
VRML	Virtual Reality Modeling Language
VTC	Visual Texture Coding
WSS	Wavelet Subdivision Surface
WWW	World Wide Web
X3D	eXtensible 3D
XML	eXtensible Markup Language
XMT	eXtended MPEG-4 Textual format

Context and Objectives

With the development of the computer networks, mainly the Internet, it became easier to develop applications where the execution is shared between a local computer, the Client, and one located on the other side of the network communication channel, the Server. The so called distributed applications significantly changed the way computing is done because, by offloading some part of the work to the Server, the client terminal can be less powerful, hence less expensive. However, distributed applications are not new. In fact, in the beginning of the computer development, it was the only practical way of using computers.

The development of mobile devices made computing on the go possible. Usually mobile devices have limitations in terms of computing power and storage capacity, therefore distributed applications are one of the candidate solutions. Nowadays, mobile devices are almost always connected to the Internet, thus distributed computing on mobiles is almost as easy as for a fixed terminal. Most mobile distributed applications are web based, using the HTTP protocol together with the HTML file format and JavaScript to build the application. The limitations related to these standards encapsulate the applications in a restricted environment. This is most obvious for entertainment applications, mainly games, because most of them are impossible to implement by using only these standards. Bringing Rich Internet Applications (RIA) to the mobile devices may overcome this gap, however their virtual execution environment further stresses the already scarce resources of the mobile device. Therefore, there are serious technical limitations in using distributed solution for entertainment applications. In addition, mobile applications development brings other challenges, mainly due to the mobile devices heterogeneous environment, both in terms of software and hardware.

The hardware advancements in the recent years made it possible to display 3D graphics (games, map navigation, virtual worlds) on mobile devices. However, executing these complex applications on the client terminal is not possible without reducing the quality of the displayed graphics or lowering its processing requirements. Different solutions have already been proposed in academic publications, however none of them satisfies all requirements.

The objective of this thesis is to propose an alternative solution for a new client-server architecture where the connectedness of the mobile devices is fully exploited.

Several main requirements will be addressed:

1. Minimize the network traffic and reduce data rate fluctuations,

2. Reduce the required computational power on the terminal, and

3. Preserve the user experience compared with local execution.

Organization of the thesis

Chapter I provides a short overview of the history of distributed systems. First it presents the different stages of remote computing developed since the beginning of the Internet. Then the state of the art in distributed applications with main focus on distributed 3D applications is presented. An analysis of these solutions is provided, exposing their advantages and disadvantages by taking into account three criteria: hardware requirements at the terminal side, the network bandwidth and the visual appearance. The second part of Chapter I presents an overview of different formalisms for representing scene-graphs, the core of any game, containing all visual elements, as well as their connectedness. They are compared according to their supported features, focusing on graphics elements, streaming support as well as some special requirements, like compression and interactivity support. The most appropriate scene-graph is selected and it is presented in detail.

Chapter II proposes the main contribution of this thesis. The first part introduces a formal framework that can effectively define and model distributed applications. The intention is to use the framework to analyze the previously presented architectures from a theoretical point of view, and draw conclusions about their capabilities. Then a model of new architecture is presented, that overcomes the disadvantages of the architectures presented in the state of the art. The next part explores the design of MPEG-4 capable architectures optimized for running on a PC. Some application may implement their own scene-graph structure and use MPEG-4 only for storing assets, hence implementing a complete MPEG-4 compatible player is not necessary. Therefore, an API is proposed for accessing MPEG-4 content that is easy to integrate and use in third party applications. The last part of this chapter explores the design of MPEG-4 capable architectures optimized for running on mobile devices, less powerful by nature. Several optimizations needed to adapt the architecture are presented.

Chapter III presents the experiments and the validation of the contributions proposed in Chapter II. The first part validates the formal framework by using it to model each of the architectures presented in the state of the art. The second part of the chapter validates the proposed architectures for a powerful platform as well as for mobile devices. The performance of the player for mobile devices is evaluated with a goal to find the maximum supported complexity of 3D data. Then a design of the new client-server architecture will be validated by implementing a game and running simulations.

The last chapter concludes the work and proposes some possible extensions.

Resumé long

I.1 Introduction

Avec le développement des réseaux informatiques, principalement d'Internet, il devient de plus en plus facile de développer des applications dont l'exécution est répartie entre un ordinateur local, le client, et un ordinateur à distance (à une autre extrémité du canal de transmission), le serveur. Ainsi, ces applications dites applications distribuées ont significativement modifié la façon d'effectuer les calculs puisqu'en sous-traitant une partie de cette tache au serveur, le terminal client nécessite moins de puissance et est par conséquence moins cher. Cependant, les applications distribuées ne sont pas vraiment nouvelles. En réalité, aux débuts de l'informatique, elles étaient le seul moyen d'utiliser les ordinateurs.

Le développement des appareils mobiles a généralisé l'usage de l'informatique. Habituellement, les appareils mobiles sont limités en termes de puissance de calcul et de capacité de stockage. Dans ce cas, les applications distribuées sont une solution intéressante. A l'heure actuelle, les dispositifs mobiles sont presque toujours connectés à Internet, ainsi le calcul distribué peut ce faire presqu'aussi facilement qu'avec un terminal fixe. La plupart de ces applications sont orientées web et font appel au protocole HTTP, au format de fichier HTML et à JavaScript dans leur mise en œuvre. Les limitations propres à ces standards imposent un environnement restreint à ces applications. En particulier, pour les applications à but récréatif, principalement les jeux, il est impossible de les mettre en œuvre en utilisant uniquement ces standards. L'arrivée des applications Internet riches devrait combler ce fossé, cependant leur environnement d'exécution virtuel est consommateur en ressources alors que celles-ci sont déjà rares dans un environnement mobile. Par conséquent, les solutions distribuées pour les applications de divertissement doivent faire face à des limitations techniques importantes.

Par ailleurs, le développement d'application pour mobiles pose de nombreux autres problèmes, principalement à cause de l'hétérogénéité des environnements, à la fois logiciels et matériels.

Les progrès techniques de ces dernières années au niveau matériel ont rendu possible l'affichage en 3D (jeux, navigation cartographique, mondes virtuels) sur les mobiles. Cependant, l'exécution de ces applications complexes sur le terminal client est impossible, à moins de réduire la qualité des images affichées ou les besoins en calcul de l'application. Différentes solutions ont déjà été proposées dans la littérature mais aucune d'entre elles ne

satisfait l'ensemble des besoins. L'objectif de cette thèse est de proposer une solution alternative, c'est à dire une nouvelle architecture client-serveur dans laquelle l'interconnexion des dispositifs mobiles est complètement exploitée.

Les principales conditions de mise en œuvre seront traitées:

1. Minimiser le trafic réseau,
2. Réduire les besoins en puissance de calcul du terminal, et
3. Préserver l'expérience utilisateur par rapport à une exécution locale.

I.2 Evolutions récentes des réseaux informatiques

L'ère de l'informatique moderne à débuté suite à la période d'expansion d'internet avec l'émergence des applications distribuées. Dans cette partie nous analysons les aspects les plus importants des solutions modernes d'informatique en réseau en commençant par une brève chronologie de l'évolution des architectures pendant ces 20 dernières années. Les premières applications ne fournissaient que des services limités comme l'accès à différents types de données. Le développement d'Internet, en apportant plus de fonctionnalités avancées et des applications complètes, a permis l'émergence de systèmes plus complexes. Le développement ultérieur d'Internet et des terminaux a permis d'utiliser ces derniers pour faire du traitement et ainsi contribuer à la création d'applications mixtes qui traitent une partie des données sur le serveur et une autre partie sur le client.

I.2.1 Une nouvelle approche dans le traitement des applications multimédia

Les applications complexes comme certains jeux nécessitent une importante puissance de calcul qui n'est pas toujours disponible avec des terminaux légers comme les téléphones mobiles et les PDAs. En utilisant l'interconnexion de ces dispositifs, la tendance de ces dernières années est d'avoir des applications client-serveur légères dans lesquelles le serveur effectue les calculs complexes et le client ne gère alors que l'interaction et la visualisation. Ainsi le paradigme évolue vers des clients légers et des serveurs puissants. Le client léger est un dispositif mobile avec une connexion à Internet théoriquement omniprésente. Le fait d'avoir un système de serveurs centralisés permet aussi d'accrotre l'efficacité de la solution en terme de puissance. Avoir un serveur en charge la majorité du temps est beaucoup plus efficace, du point de vue énergétique, que d'avoir une multitude de PCs qui sont le plus souvent en attente.

L'industrie du divertissement, en particulier celle du jeu, est l'une des plus importantes à soutenir le développement des PCs. Il y a toujours une course poursuite entre les producteurs de jeux et les constructeurs d'ordinateurs. Les premiers essayent de produire de meilleures représentations visuelles tandis que les seconds s'attachent à fournir de meilleures performances matérielles qui puissent satisfaire les besoins de ces jeux. Pour tester un nouveau jeu, il est souvent nécessaire de mettre à jour son matériel avec un impact important sur le budget des utilisateurs. Malgré ce cot, ce matériel ne sera finalement

utilisé que de façon sporadique et donc cet investissement sera peu efficace. Une des solutions réside dans l'exécution du jeu par un système centralisé où les clients se connectent au serveur pour jouer [43]. OnLive[1] en est un exemple : ce type de service permet de jouer à distance à des jeux récents. L'utilisateur n'a besoin que d'un PC d'entrée de gamme avec un navigateur internet et d'une connexion Internet avec une bande passante appropriée. Le principe du service est le suivant: le rendu du jeu s'effectue sur une machine puissante et un flux audio-vidéo est envoyé au client. De l'autre cté, le client envoie les commandes du jeu au serveur. Ainsi, le client n'effectue que du décodage audio et vidéo. Certains boitiers vendus par les FAI peuvent aussi fournir ces services. L'inconvénient de ce système est qu'il ne peut pas fonctionner correctement dans un environnement sans fil. Ceci est du à une forte sensibilité à la latence et à la gigue. Bien que de nombreux appareils mobiles puissent décoder et afficher la vidéo en temps réel, la connexion sans fil n'est pas assez stable pour assurer une qualité constante. Aussi, d'autres architectures doivent tre utilisées pour les terminaux mobiles; ceci constitue l'objet de ce travail de recherche.

I.3 Analyse des architectures distribuées pour la 3D et les jeux

La figure I.1 présente les différentes architectures analysées dans l'état de l'art. Les méthodes sont ordonnées de la gauche vers la droite en fonction de leur dépendance à un serveur. On distingue trois catégories: (1) la logique du jeu et les données sur le client, (2) la logique du jeu sur le client et les données sur le serveur, et (3) la logique du jeu et les données sur le serveur.

A l'extrme gauche, se trouvent les techniques pour lesquelles toute l'application s'exécute sur le client alors qu'à l'extrme droite se trouvent les techniques avec une exécution intégralement déportée sur le serveur et un affichage sur le client.

Les techniques sont séparées en deux groupes principaux: les techniques pour lesquelles la logique de l'application s'exécute sur le client et celles pour lesquelles la logique s'exécute sur le serveur. Le premier groupe est séparé en deux sous-parties: les techniques où l'on stocke les données sur le client et celles où les données sont sur le serveur.

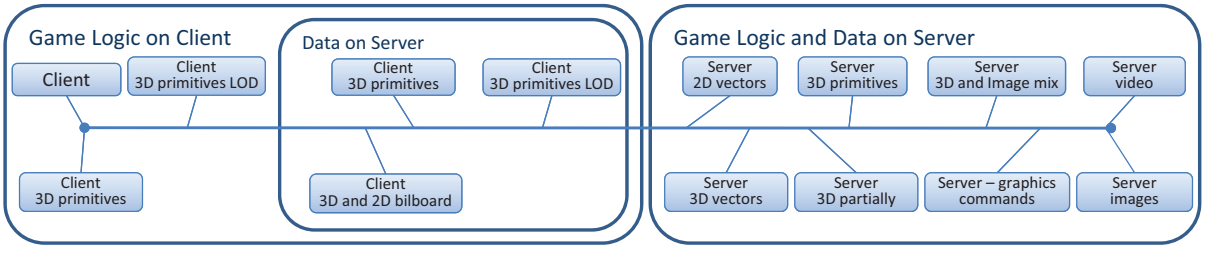


Figure I.1: Techniques pour l'affichage 3D

L'analyse de la figure I.1 amène plusieurs conclusions. D'après l'état de l'art effectué par Capin et al. dans [20], de nombreuses solutions existent pour optimiser le rendu sur

[1] www.onlive.com

les terminaux mobiles, cependant elles sont toujours limitées par la puissance de calcul de ces dispositifs. Par exemple, le développement d'un très beau jeu d'échecs nécessiterait de diminuer les capacités du jeu afin de permettre le traitement graphique. Capin et al. en concluent que le rendu à distance peut tre une solution viable, et une solution dans laquelle le traitement s'équilibre entre les approches locales et déportées constitue une piste de recherche intéressante. Ainsi, cette thèse poursuit dans cette direction et se concentre sur les architectures où la logique du jeu s'exécute sur le serveur et le rendu s'effectue sur le client.

Les architectures peuvent tre séparées en six catégories principales:

1. Commandes graphiques
2. Pixels 2D
3. Primitives 2D
4. Vecteurs 3D
5. Objets 3D simples
6. Objets 3D multiples

Toutes ces architectures ont été analysées et il a été conclu que:

- Les techniques qui utilisent des transferts d'images ou de commandes graphiques ne sont pas adaptées aux réseaux mobiles à cause de leur besoins en bande passante,
- Les techniques qui transfèrent des primitives graphiques 2D ou 3D sont meilleures, mais elles manquent d'un contrle approprié sur les données dans la mesure où il n'y a pas de graphe de scène pour organiser les données.
- Les techniques qui convertissent les données en lignes (2D ou 3D) ne sont pas visuellement satisfaisantes.

Ainsi, aucune des architectures ne satisfait simultanément l'ensemble des besoins et donc une architecture alternative, qui peut tre appliquée aux jeux sur mobiles, est nécessaire. Il a été observé que l'organisation du graphe de scène a une importance majeure pour les programmes de type jeux. Par conséquent, la partie suivante décrit l'état de l'art des différents formats de graphe de scène.

I.4 Analyse de formats de graphe de scène

Les formalismes de graphe de scène suivants ont été analysés: VRML, X3D, SMIL, SVG, COLLADA, MPEG-4 BIFS, XMT et LASeR.

Il a été observé que chacun d'entres eux a été mis au point pour des applications différentes. Le choix d'un standard dépend de sa capacité à supporter ces trois contraintes:

- 3D: tre capable de représenter des objets en 3D et de les animer

- Compression: réduire la taille des données transmises
- Streaming: avoir la possibilité de commencer à utiliser le contenu avant qu'il ne soit complètement téléchargé

Le tableau I.2 résume les caractéristiques de tous les standards présentés précédemment. Notons que SMIL et SVG ne supportent pas la 3D, ni le streaming, ni la compression. VRML et X3D supportent la 3D mais pas le streaming ni la compression. Bien que COLLADA soit très bon en tant que format interchangeable, il lui manque le support du streaming et de la compression. Comme on peut l'observer dans le tableau I.2, il n'y a que le standard MPEG-4 qui satisfasse toutes les contraintes en étant capable de gérer la 3D, le streaming, la compression et la possibilité de mettre la scène à jour pendant l'exécution.

Table I.2: Comparaison des standards pour la description des scènes multimédia

Caractéristiques Supportées:	VRML	X3D	SMIL	SVG	COLLADA	MPEG4		
						BIFS	XMT	LASeR
Primitives:								
Texte	X	X	X	X		X	X	X
2D			X	X		X	X	X
3D	X	X			X	X	X	
Audio	X	X	X			X	X	X
Vidéo	X	X	X			X	X	X
Animation	X	X	X	X	X	X	X	X
Streaming:								
2D						X	X	X
3D						X	X	
Audio	X	X	X			X	X	X
Vidéo	X	X	X			X	X	X
Animation						X	X	X
Synchronisation	X	X	X			X	X	X
Special:								
Compression						X	X	
Interactivité	X	X				X	X	X
Evènement (client)						X	X	X

I.5 Cadre formel pour les systèmes 3D distribués et le design d'architectures de lecteurs MPEG-4

Afin d'analyser les architectures présentées dans l'état de l'art, nous définissons tout d'abord une représentation théorique. Pour ce faire, les systèmes sont divisés en plusieurs

blocs de traitement connectés les uns aux autres et qui transforment les données en entrée. Ces blocs doivent tre choisis avec attention afin de pouvoir modéliser tous les systèmes qui ont été présentés. Le modèle va aider à mieux comprendre les opérations et les scénarios d'utilisation, de mme que leurs limitations.

I.5.1 Définition du cadre formel

La chane de traitement d'un système distribué peut tre représentée comme un flux d'informations faisant appel à des transformations. La base de ce travail a été réalisée précédemment par Preda et al. dans [56], où ils ont analysé des systèmes de visualisation de modèles graphiques 3D, pouvant provenir d'un serveur à travers un canal réseau. Une limitation de ce travail est que la modélisation ne prend en compte qu'un seul objet 3D, alors que des applications distribuées complexes, ex. les jeux, intègrent de nombreux objets 3D et d'autres médias de différentes complexité.

Dans le modèle mathématique proposé dans [56], chaque objet est représenté par un ensemble de caractéristiques $\{F_i\}$ qui sont les entrées de chaque fonction de base. Pour adapter des applications plus complexes, au modèle présenté dans ce chapitre, l'entrée de la fonction de traitement est le graphe de scène Sg, qui est défini comme un ensemble de nœuds $\{N_i^t\}$ au temps t. Les nœuds peuvent tre des groupes de nœuds Nd qui contiennent des données de rendu. Le graphe de scène est défini dans l'équation 1.

$$Sg = \left\{N_i^t\right\} = \left\{Ng_i^t, Nd_i^t\right\} \tag{1}$$

La sortie du modèle reste la mme, $\{P_i\}^{2D} = \{R_i, G_i, B_i, A_i\}$, un ensemble de pixels 2D, avec les composantes RGB et la transparence (A), prts à tre affichés; ou $\{P_i\}^{3D} = \{R_i, G_i, B_i, A_i, D_i\}$, un ensemble de pixels 3D contenant une composante de profondeur calculée au moment du rendu. Le but est de produire soit un, soit deux ensembles $\{P_i\}^{2D}$ ou $\{P_i\}^{3D}$ à partir du graphe de scène SG^t de façon optimisée, en prenant en compte les contraintes inhérentes aux composantes du systèmes.

Les transformations possibles dans le processus sont: le rendu, le codage, la simplification, la modélisation et la mise à jour des scènes. Leur définition mathématique est la suivante:

$$Rendering : \{P_i\} = R\left(Sg\right) \tag{2}$$
$$Coding : \left\{Sg_i^C\right\} = C\left(Sg\right) \tag{3}$$
$$Simplification : \left\{SG_i^S\right\} = S\left(Sg\right) \tag{4}$$
$$Modeling : \{Sg_i\} = M\left(Sg\right) \tag{5}$$
$$Scene - updates : Sg^t = U\left(Sg^{t-1}, \{I_i\}\right) \tag{6}$$

Où $\{I_i\}$ représente les différentes entrées du jeux.

En utilisant le cadre formel précédemment décrit, la section suivante analyse les techniques qui représentent l'état de l'art d'un terrain ordinaire.

I.5.2 Analyse du cadre formel

La section I.5.1 propose un cadre formel pour décrire des architectures de systèmes distribués pour des rendus graphiques 2D et 3D. En utilisant ces transformations, quelques architectures peuvent tre décrites et définies. On peut en conclure que les transformations sont suffisantes pour décrire des architectures distribuées. Cependant, ce formalisme n'est pas limité à cette catégorie d'architecture, mais au contraire, l'intention est de pouvoir l'utiliser avec n'importe quelle architecture distribuée qui sera développée dans le futur.

Cette thèse propose une architecture pour jouer à des jeux 3D complexes sur téléphone mobile. Etant donné que les téléphones mobiles ont une capacité de traitement assez réduite, les architectures qui se basent sur une représentation à base de pixels ne sont pas appropriées et la bande passante nécessaire pour satisfaire l'expérience est importante. Néanmoins, les dispositifs mobiles ont la capacité de traiter des rendus graphiques 2D et 3D et il est donc nécessaire d'utiliser d'autres architectures. les architectures à base de commandes graphiques ont besoin du mme équipement graphique que les PC, ceux-ci n'étant pas disponibles pour téléphones mobiles, ils ne peuvent pas tre utilisés. Les architectures basées sur des primitives 2D nécessitent moins de bande passante en envoyant uniquement des vecteurs 2D au client. Toutefois, il est nécessaire de les envoyer pour chaque image et leur apparence n'est pas satisfaisante dans un jeu. Les architectures basées sur des vecteurs 3D ont une bande passante plus efficace en envoyant des vecteurs 3D une seule fois pour chaque objet, mais leur apparence physique reste similaire à celle des primitives 2D. L'architecture où chaque objet est transféré ne supporte pas des applications avec des graphes de scènes complexes, mais elle peut tre utilisée comme une partie de l'architecture. Ainsi, les architectures multi objets semblent les plus appropriées, étant capables de rendre des images visuellement satisfaisantes et de supporter des graphes de scène complexes. Néanmoins, une de leurs limitations est que l'utilisateur peut seulement contrler une proportion de la caméra ce qui ne permet pas son utilisation pour les jeux où les actions des utilisateurs sont plus sophistiquées.

I.5.3 Conception d'une nouvelle architecture de systèmes distribués

L'architecture modelée correspond à une architecture distribuée qui est capable de supporter des applications de jeux pour des dispositifs mobiles. En utilisant le cadre formel, l'équation de l'application peut tre définie comme suit:

$$\{P_i\} = R \circ C \circ S \circ U \left(Sg^{t-1}, \{I_i\}\right) \tag{7}$$

Les transformations sont réalisées dans l'ordre suivant:

1. Le graphe de scène est mis à jour par **U**
2. La structure du graphe de scène est simplifié (i.e. les noeuds sont éliminés temporellement pour la transmission) en **S**, en fonction de différentes conditions et paramètres.
3. Le graphe de scène est compressé et transmis au client par **C**

4. Le graphe de scène est rendu sur le client par **R**

Etant donné que dans tous jeux le contenu est organisé dans une espèce de graphe de scène, il est logique de conclure que le fait d'utiliser le mme pour tous les jeux augmente l'efficacité de leur développent. Le standard MPEG-4 a été proposé comme la caractéristique la plus complète pour une architecture de jeux, et sera donc utilisée comme le format de graphe de scène pour l'architecture distribuée proposée.

La deuxième standardisation peut tre faite sur le genre de données envoyées par client au serveur. Par exemple, dans une course de voitures, le fait d'appuyer sur la touche "avance" incrémente la variable d'accélération, mais dans un autre jeu la mme touche peut uniquement changer la position de la vue pour une valeur constante et par conséquent il est nécessaire de standardiser les données. La solution la plus facile et la plus évidente est de ne pas envoyer les actions que les touches produisent, mais le code de la touche en lui-mme. Ainsi, le serveur pourra l'interpréter et produire l'effet désiré. Ceci soulage l'application client de n'importe quel calcul pour un jeu spécifique et permet donc son utilisation dans plusieurs jeux.

Le fait de standardiser les entrées et les graphes de scène permet l'indépendance du client pour un jeu spécifique. Dans la section I.4 il a été conclu que le standard MPEG-4 avait les caractéristiques nécessaires et donc le lecteur proposé utilisera ce standard. Toutefois, toutes les fonctionnalités du standard ne pas doivent tre prises en charge pour implémenter l'architecture du jeu.

Le schéma de l'architecture distribuée proposée est présenté dans la figure I.2.

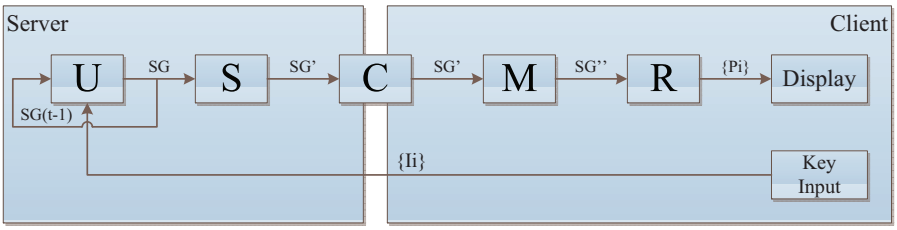


Figure I.2: Architecture proposée

Pour le rendu du graphe de scène, les nœuds qui nécessitent d'tre supportés, sont présentés dans le tableau I.3. La description détaillée des nœuds est présente dans le standard MPEG-4. La réduction du nombre de nœuds aide à l'optimisation du moteur de rendu, ainsi qu'à la réduction de la taille du code, et par conséquent la réduction du fichier de l'application. De plus, cela permet aussi de développer les applications plus rapidement, la courbe d'apprentissage du standard MPEG-4 étant raccourcie. Un nombre plus réduit de nœuds permet une optimisation du cté serveur, et accélère donc la génération de graphes de scène.

Chaque composant de l'architecture distribuée peut tre associé avec la technologie correspondante. Le graphe de scène est basé sur le format de graphe de scène MPEG-4, i.e. BIFS. La scène met à jour le composant **U** qui dépend du jeu en lui-mme et l'entrée est reçue à partir du client en utilisant une requte AJAX (i.e. HTTP). La simplification du composant **S** permet d'éliminer les nœuds qui ne sont pas visibles dans la position actuelle de la caméra. Le composant de codage traite les mises à jour du graphe de scène

Table I.3: Sous-ensemble de nœuds BIFS

Node Name	Node Name
Appearance	Text
Background2D	TextureCoordinate
Coordinate	Transform
FontStyle	Transform2D
Group	Valuator
ImageTexture	Viewpoint
IndexedFaceSet	InputSensor
Inline	BitWrapper
Layer2D	SBBone
Layer3D	SBSegment
Material	SBSite
Material2D	SBSkinnedModel
NavigationInfo	SBVCAnimation
Normal	MorphShape
OrderedGroup	SBVCAnimationV2
Shape	Rectangle
Script	Switch

et génère les commandes d'actualisation BIFS qui sont transmises au client à travers le réseau en utilisant RTSP, où ces commandes sont alors décodées et appliquées à la scène BIFS. Le composant de rendu **R** décompose la scène BIFS et l'affiche.

I.5.4 Conception d'une Architecture de Lecteur MPEG-4 pour Plateformes Puissantes

Dans cette section nous présentons un lecteur MPEG-3 de contenu 3D qui a été conçu pour optimiser le modèle SDM (System Decoder Model) MPEG-4 et qui utilise la capacité de traitement multi-core pour obtenir un meilleur rendu et une expérience utilisateur plus fluide et plus rapide. De plus, il est possible de faire évoluer ses composantes pour inclure de nouveaux formats graphiques.

Etant donné que le lecteur est basé sur le standard MPEG-4, il est logique d'en dériver les conditions nécessaires à son bon fonctionnement, mme s'il ne doit pas contenir toutes ses caractéristiques.

Les conditions nécessaires qui en dérivent sont les suivantes:

1. Réception des données à partir de plusieurs sources : fichier ou réseaux
2. Utilisation optimale de la capacité du client terminal
3. Synchronisation de tous les flux de données
4. Adaptation de multiples formalismes de graphe de scène
5. Intégration facile de différents codeurs

6. Intégration facile de différentes applications

La première condition est satisfaite par le SDM original en implémentant l'interface DMIF.

L'implémentation doit tre faite dans l'architecture du lecteur puisque la deuxième condition n'est pas spécifiée par le SDM.

La troisième condition n'est pas complètement satisfaite par le SDM original. Alors que la synchronisation entre les flux de vidéo et audio n'est pas très complexe, celle des rendus de scènes 3D connat plus de difficultés dues à la quantité de ressources nécessaires. Celles-ci incluent les données de mesh, de textures et d'animation. Toutes ses ressources doivent tre chargées dans la carte graphique, et pour le faire de façon optimale il est nécessaire d'utiliser un seul fil d'exécution pour le processus de chargement. De plus, le mesh et ses données d'animation sont très liées dans le sens où le mesh ne peut pas tre rendu sans ces données.

La quatrième condition n'est pas satisfaite par le SDM car il utilise seulement un formalisme de scène graphique (i.e. BIFS).

En ce qui concerne la cinquième condition, le SDM spécifie chaque décodeur comme un composant indépendant; cependant, il ne spécifie pas l'interface d'intégration. Dès lors, c'est à l'implémentation de l'architecture du lecteur de le concevoir.

La sixième condition n'est pas satisfaite par la conception SDM originale.

Le tableau I.4 résume les conditions antérieurement décrites et comment elles sont résolues.

Table I.4: Conditions nécessaires d'un lecteur MPEG-4 pour PC

Conditions nécessaires	**Solution**
Réception de données à partir de plusieurs sources, fichier ou réseaux	Définition de l'interface standard pour différent streams d'entrée
Synchronisation de tous les streams de données	Implémentation du composant "Timer" commun
Adaptation de multiples formalismes de graphe de scène	Définition de l'intermédiaire de graphe de scène
Intégration facile des différents décodeurs	Définition d'une interface standard pour l'intégration des décodeurs
Intégration facile dans différentes applications	Prédéfinition des fonctions et des messages pour communiquer avec l' architecture du lecteur
Utilisation optimale de la capacité du terminal client	Séparer les différentes tches en plusieurs fils d'exécution

I.5.5 MPEG Extensible Middleware (MXM)

Accéder à du contenu MPEG-4 nécessite une profonde connaissance de ce standard. Bien qu'il existe des systèmes pour accéder à du contenu MPEG-4 qui incluent des logiciels de référence et une implémentation par un tiers comme GPAC, il n'est pas aisé de développer un logiciel MPEG-4 performant. En conséquent, le fait d'avoir une API simplifiée peut tre

d'une grande importance pour la diffusion de l'utilisation du standard. Le rle du MPEG Extensible Middleware (MXM) est de s'adapter exactement au problème. Nous avons contribué au processus de standardisation du MXM dans les aspects graphiques 3D.

L'API Graphics3D donne accès à du contenu 3D des données MPEG-4. elle décompose les scènes et convertit chaque mesh dans une représentation intermédiaire qui est suffisamment simple pour une décomposition par une application externe. L'API est divisée en trois parties, chacune spécialisée dans différentes parties du contenu:

- Apparence - API pour l'apparence du mesh
- Géométrie - API pour le contenu 3D mesh
- Animation - API pour l'animation du mesh

I.5.6 Conception d'une Architecture de lecteur MPEG-4 pour Dispositifs Mobiles

Plusieurs implémentations graphiques de MPEG-4 sont accessibles dans des produits comme ceux proposés par iVast ou Envivio ou dans des paquets open-source comme GPAC [33]; cependant la littérature concernant les graphiques 3D pour MPEG-4 mobile est quasi inexistante. Pour quantifier les capacités de qualité graphique utilisée dans un jeux, un lecteur graphique 3D de MPEG-4 a été conçu et implémenté pour une plateforme Nokia S60 basé sur un Symbian S60 FP1 SDK [30]. L'équipement testé inclut le Nokia N93, le Nokia N95 et le Nokia N95 8GB (ils ont presque les mmes caractéristiques et les mmes performances). Pour s'assurer de bonnes performances, le lecteur a été implémenté dans les langages C et C++. Pour le démultiplexage et le décodeur BIFS, l'implémentation est totalement faite au niveau logiciel (basé dans le cadre GPAC). L'implémentation du rendu est supportée par l'équipement (basé sur OpenGL ES [63]).

Pour concevoir d'une façon adéquate le lecteur mobile MPEG-4, il est nécessaire de réaliser une analyse des équipements mobiles et d'établir une liste de conditions nécessaires:

- Simplifier l'architecture
- Trouver le codeur approprié pour les dispositifs mobiles
- Réduire le nombre de nœuds BIFS
- Supporter des entrées à partir de multiples flux

I.6 Expérimentations et Validation

I.6.1 Lecteur MPEG-4 pour Dispositifs Mobiles

Cette section présente les composants du cté client d'une architecture de systèmes distribués présentée dans le chapitre précédent: Le composant de rendu (**R**) et une partie du composant codage (**C**). Comme il a été illustré dans la figure I.2, le composant codage

est séparé entre le serveur et le client. Dans le client, les données transmises sont tout d'abord décodées et ensuite rendues. Ainsi, un des objectifs de notre expérimentation est de trouver les limites supérieures en termes de complexité des caractéristiques 3D (en ce qui concerne la géométrie, la texture et l'animation) en assurant un décodage rapide. Le deuxième objectif est d'analyser les performances de rendu pour la plateforme choisie.

Le premier test concernant des fichiers MPEG-4 contenait exclusivement des objets statiques avec un nombre différent de sommets et de triangles. Le décodage BIFS a été mesuré, ainsi que le taux (images par seconde) de rendu pour chaque fichier. Le deuxième test correspond à des objets animés basés sur l'approche de déformation "skeleton-driven". Ce type de contenu, ayant besoin de plus de calculs dus aux opérations réalisées durant l'animation (par sommet), il entraine un nombre d'images par seconde pour le rendu inférieur à celui des objets statiques.

En ce qui concerne les capacités de décodage MPEG-4 et les capacités de rendu du N95, on peut observer que pour obtenir un temps de décodage acceptable (moins de 2 secondes), un objet 3D représenté en MPEG-4 doit avoir moins de 58 000 primitives de bas niveau (qui correspondent à des objets avec moins de 15 0000 sommets et 12 0000 triangles) pour des objets statiques et 1 500 triangles pour des objets animés. Le N95 est capable de fournir un rendu d'objets texturisés et de luminosité statique d'environ 20 000 sommets et 20 000 triangles avec un taux acceptable de 25 images par seconde et il est possible d'avoir des rendus avec ce mme taux pour des objets animés texturisés et éclairés d'environ 6 000 sommets et 6 000 triangles.

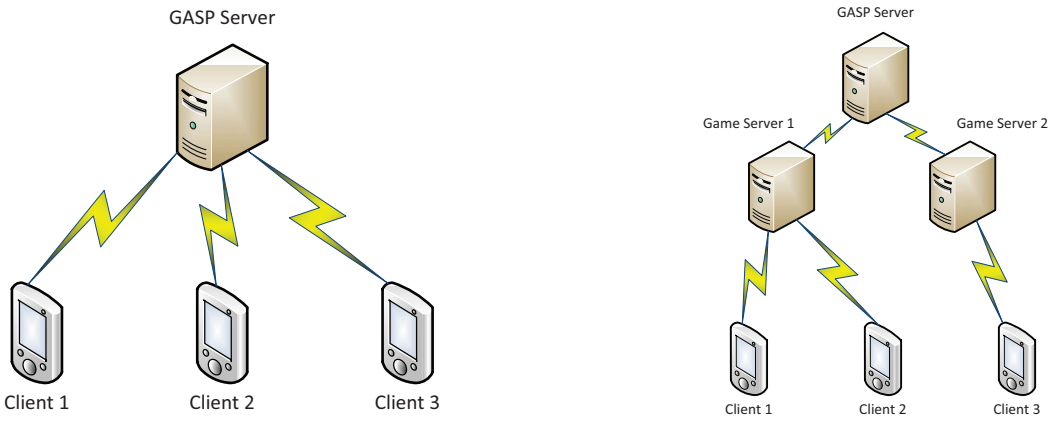
La taille réduite de l'affichage sur les téléphones mobiles implique l'utilisation d'un nombre limité de pixels pour le rendu. La simplification de la géométrie et de la texture peut tre utilisée sans affecter la perception visuelle des objets. De cette façon, la taille du contenu peut tre réduite de 60-80% et le temps de chargement de 70-90% en accord avec Preda et al. [56]. La taille de la géométrie est réduite d'après la technique d'erreur quadratique de Garland [35], et la taille de la texture en réduisant la largeur et la longueur. Le fait de réduire la taille du fichier MPEG-4 à 428kb (24% de la taille originale) réduit le temps de chargement à 3,8 secondes (57% plus rapide que l'original).

I.6.2 Validation de l'Architecture à partir d'un Jeux

Basé sur l'architecture proposée dans la section précédente, un jeu multi-joueurs de course de voitures a été implémenté. Les conditions du jeu changent fréquemment et il est donc approprié pour tester notre architecture. Le jeu a été originellement développé en J2ME comme un jeu multi-joueurs traditionnel pour téléphone mobile. L'utilisateur contrle uniquement la vitesse, cependant il y a deux paramètres supplémentaires qui affectent la vitesse et les capacités de freinage: l'endommagement des pneus et/ou des freins. l'endommagement des freins augmente chaque fois que la voiture réduit sa vitesse et celui des pneus augmente chaque fois que la voiture passe un virage avec une vitesse supérieure à celle recommandée.

Le jeu utilise le serveur GASP [3] pour communiquer entre les joueurs. Originellement, le moteur logique et celui de rendu étaient implémentés dans le logiciel J2ME; le serveur GASP était utilisé seulement pour transmettre la position entre les joueurs et un moteur

de rendu 2D très simple était utilisé pour créer les pistes et les voitures [1]. L'architecture du jeu est illustrée dans la figure I.3a.



(a) Architecture Originale (b) Architecture Adaptée

Figure I.3: l'architecture originale du jeu de courses (a) et l'architecture adaptée (b)

L'adaptation du jeu a été faite en plusieurs étapes :

1. Identification des structures utilisées pour garder les données pertinentes (position des voitures, et rotation de la camera pendant la course).
2. Identification de la boucle principale et des appels de rendu.
3. Rajout des composantes de communication dans la boucle principale, du jeu au lecteur et vice-versa.
4. Définition des messages de commande BIFS, codification, encapsulation et analyse du message (pour envoyer la position des voitures aux joueurs et recevoir les commandes des joueurs respectivement).
5. Conversion de tous les composants et du graphe de scène du jeu en fichiers MPEG-4.

I.6.2.1 Experiences et Résultats

Plusieurs expérimentations ont été mises en place pour mesurer objectivement l'expérience des utilisateurs au moment de jouer, basées sur les temps de réponse et d'interaction (phases 1 et 3) et sur le temps d'attente pour la transmission et le chargement des composants 3D (phase 2).

Le tableau I.5 montre la latence quand les composants sont transmis et la figure I.4, la latence quand seulement l'interaction de l'utilisateur et les commandes d'actualisation sont transmises. Les mesures ont été réalisées pour deux configuration réseaux différentes : Wi-Fi (IEEE 802.11, ISO/CEI 8802-11) et UMTS.

En ce qui concerne la connexion Wi-Fi, le temps moyen d'exécution est de 80 ms avec un maximum de 170 ms. Dans le cas d'une connexion UMTS le temps moyen est de 350 ms avec un maximum de 405 ms. Ces résultats correspondent à la boucle entière, c'est-à-dire le temps de la transmission des interactions utilisateurs, celui du traitement par le

Table I.5: Latence (transmission et décodage) pour des composants 3D utilisé dans le jeu de courses

Composant	Voiture_v1	Voiture_v2	Circuit_v1	Circuit_v2
Nombre de sommets	253	552	1286	7243
Taille du fichier MPEG-4 (KB)	82	422	208	1600
Temps de transmission Wi-Fi (ms)	27	126	68	542
Temps de transmission UMTS (ms)	422	2178	1067	8246
Temps de décodage (ms)	112	219	328	2538
Temps d'attente total Wi-Fi (ms)	139	345	396	3080
Temps d'attente total UMTS (ms)	534	2397	1395	10784

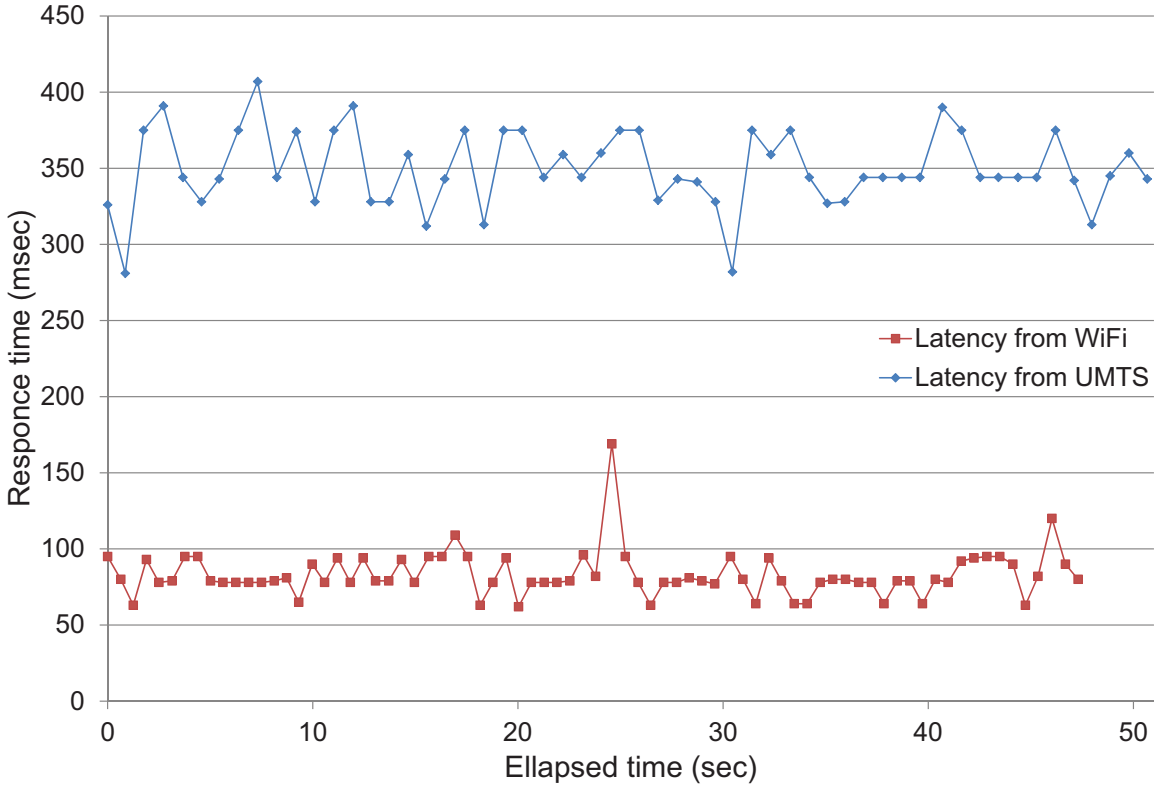


Figure I.4: Temps de réponse (en ms) pour UMTS et Wi-Fi enregistré pendant la phase 3 "Jouer le jeu". L'axe horizontal la durée du jeu

serveur, la transmission des commandes BIFS, le décodage et le rendu de l'actualisation des scènes locales.

I.6.3 Système d'Animation en Ligne

Afin de valider l'architecture du lecteur pour une plateforme puissante, un système en ligne pour le langage des signes a été conçu. Ce système est capable de synthétiser en temps réel des animations du visage et des mains pour le langage des signes, basé sur un texte ou un discours saisi par l'utilisateur. La nouveauté de cette approche consiste à utiliser une architecture de système basée dans un serveur dédié capable de réaliser des opérations couteuses et d'obtenir des résultats de rendu comme les flux d'animation compressés. Du cté de l'utilisateur, il suffit d'avoir un lecteur de graphiques 3D: il est tout à fait possible d'implémenter l'application dans des terminaux très légers. Cette approche a l'avantage de traiter le discours, en évitant des pertes de qualité de la voix dues à des erreurs de transmission ou de variations de bande passante. L'architecture proposée est illustrée dans la figure I.5. Du cté du serveur, les deux entrées, voix et texte, sont converties dans des paramètres d'animation. Ces derniers sont codés comme des flux d'animations MPEG-4 et diffusés sur le réseau. Cté client, un lecteur MPEG-4 reçoit le flux de l'animation et fait les mises à jour de façon continue, le graphe de scène définit le visage et la main de l'avatar.

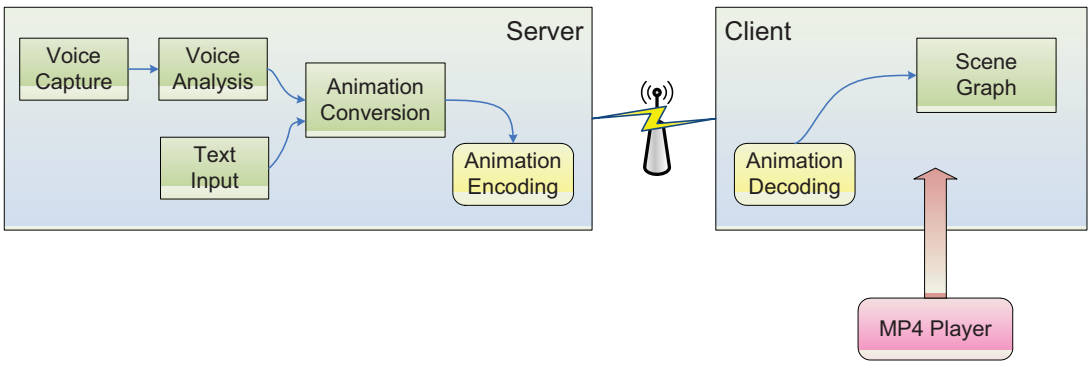


Figure I.5: Architecture proposée pour un système en ligne de langage des signes

I.6.4 MPEG Middleware Extensible (MXM)

Dans la section I.5.5 une API pour accéder à du contenu MPEG-4 était proposée en utilisant seulement quelques fonctions et sans aucune connaissance du standard MPEG-4. Dans le but de valider l'API, un visionneur de fichier MPEG-4 a été implémenté en utilisant le moteur de rendu Ogre 3D [2]. Le moteur Ogre 3D a son propre format pour stocker les données et les matériels du mesh 3D mais propose des structures complètes pour charger des données d'autres formats.

[2]http://www.ogre3d.org/

I.7 Conclusion et Perspectives

L'objectif principal de cette thèse est de proposer une architecture client-serveur alternative de pour créer des jeux sur mobile où la connectivité des dispositifs mobiles est utilisée. Les nouveaux jeux en 3D nécessitent beaucoup de puissance de calcul, aussi bien générale (CPU), que spécialisée (GPU). Les diapositifs mobiles ne sont pas capables de tout exécuter de façon efficiente, d'où la nécessité d'avoir une application qui fasse l'équilibre. Ainsi, une solution distribuée a été proposée : la logique du jeu est exécutée par le serveur et le rendu est exécuté par le client. Pour que cette solution soit efficiente, trois conditions sont nécessaires:

1. Minimiser le trafic réseau,
2. Exploiter de façon optimale les performances du terminal, et
3. Retenir l'expérience de l'utilisateur comme si le jeu était exécuter localement

Dans un premier lieu, le développement des architectures d'équipements distribuées ont été analysés, accompagnées de la présentation des avancements relatifs aux calculs à distance en mettant un accent dans les applications internet. A notre connaissance aucune des architectures ne satisfait toutes les conditions, dès lors nat la nécessité de trouver une nouvelle solution.

Il a été observé qu'une solution possible était de standardiser un composant qui soit commun à tous les jeux, à savoir le graphe de scène. En utilisant le format de graphe de scène standard il est possible de standardiser l'application client, ainsi que le protocole de communication. Par conséquent, nous avons analysé et comparer différents standards. Les analyses se sont centrées sur les capacités à supporter les caractéristiques des graphiques 3D, du streaming, de la compression et des interactions utilisateurs. Il s'est avéré que le format MPEG-4 a été le seul qui puisse intégrer toutes les caractéristiques nécessaires et a donc été celui utilisé dans l'architecture.

Pour analyser les architectures présentées dans l'état de l'art, nous avons défini un cadre qui représente les fonctionnalités de chaque étape du processus. Cette représentation mathématique nous a permis d'avoir une vue d'ensemble sur l'architecture par rapport aux restrictions et scenarios d'utilisation. Nous avons observé qu'aucun d'eux n'étaient appropriés pour les applications visées dans cette thèse. Ainsi, un nouveau cadre a été proposé en présentant toutes ses composantes et des solutions pour chacune. L'architecture utilise le standard MPEG-4 comme format de graphe de scène du jeu ainsi que comme protocole de communication entre le serveur et le client. Les données du client au serveur (i.e. commandes de l'utilisateur) sont transmises en utilisant des requtes AJAX.

La section suivante examine la conception d'une architecture qui correspond à un lecteur MPEG-4 sur une plateforme puissante. Il s'agit d'une optimisation du SDM MPEG-4 qui permet de charger des fichiers MPEG-4 ayant différents formalismes de graphe de scène. Aussi, le traitement par fil d'exécution est utilisé pour décoder les tches en parallèle, et permet une gestion plus facile des entrées des différents flux en mme temps. L'architecture a été validée en implémentant un lecteur MPEG-4 capable de supporter des fichiers qui incluent différents medias comme; des graphiques 3D, de l'animation, de

la vidéo et de l'audio ainsi que des applications distribuées. Une application web utilisant un serveur pour convertir du texte en audio a aussi été présentée ainsi qu'une qui affiche des animations pour le langage des signes.

Toutefois, nous avons remarqué que l'implémentation de cette architecture est complexe. Lorsqu'il s'agit d'applications qui utilisent seulement des caractéristiques de stockage et qui par conséquent n'ont pas besoin de toutes les caractéristiques du standard il est nécessaire de simplifier le mécanisme. C'est pour cela que nous avons conçu un MPEG Middleware Extensible qui est une simple API pour accéder à des fichiers MPEG-4. Nous proposons une API de moteur de graphiques 3D comme élément du moteur media. Nous avons de mme présenté un exemple de son implémentation en utilisant le moteur 3D Ogre. Ceci a permis de prouver que l'API était suffisamment simple pour tre utilisée dans des architectures tierces.

Etant donné que les dispositifs mobiles ont moins de capacités de calcul que les ordinateurs, l'architecture du lecteur nécessite d'tre modifiée pour refléter ces restrictions. Aussi, il est nécessaire de restreindre le format de graphe de scène (i.e. BIFS); le nombre de nœuds utilisés (au minimum) et les codecs supportés. Ces changements permettent d'avoir un lecteur optimal qui peut tre exécuté sur un dispositif mobile.

Nous avons réalisé deux types d'expériences: décodage et rendu pour du contenu statique et animé; les résultats ont été satisfaisants. Cependant, les jeux ont besoin, en plus du rendu, d'un traitement. Pour pouvoir supporter ces besoins, le nombre maximum de sommets rendus doit tre réduit, et par conséquent la qualité de l'image rendue est aussi dégradée. L'architecture proposée peut résoudre ce problème en exécutant la logique du jeu dans le serveur et le rendu dans le client.

Etant donné que le standard MPEG-4 inclut, non seulement une représentation de la scène, mais aussi une mise à jour des scènes, il a été utilisé pour représenter le graphe de scène du jeu, ainsi que pour définir le protocole de communication entre les deux composantes. L'utilisation d'un protocole standard se traduit par la possibilité de développer les composants indépendamment. D'une part, un serveur peut tre développé sans la connaissance du client, c'est-à-dire que plusieurs clients, pour différents dispositifs peuvent utiliser le mme serveur, i.e. jeux. D'autre part, le client peut tre développé en ne possédant aucune connaissance du jeu et dans un second temps tre optimisé pour un dispositif mobile spécifique tout en restant capable de jouer au mme jeu.

Pour mettre à l'épreuve cette architecture, nous avons adapté un jeu existant. Il s'agit d'une simple course de voitures dans laquelle on ne contrle que la vitesse de la voiture. Nous avons réalisé quelques expériences en mesurant la bande passante et le temps de réponse (temps écoulé entre l'appui sur une touche et la réception de la mise à jour du serveur) dans des réseaux UMTS et Wifi. Les résultats on été comparés avec les travaux de Claypool M. [26] en ce qui concerne les effets de latence sur des utilisateurs de jeux en ligne. En se basant sur ces résultats et dans la classification des jeux par rapport à la complexité et la latence, nous en avons déduit les conclusions suivantes:

- L'architecture est appropriée pour des jeux omniprésents dans les deux configurations réseau
- L'architecture est appropriée pour des jeux qui utilisent un avatar en troisième

personne avec une connexion Wifi et se trouve dans le seuil de tolérance avec une connexion UMTS

- L'architecture n'est pas appropriée pour des jeux qui utilisent un avatar en première personne avec une connexion UMTS et se trouve dans la limite inférieure de tolérance de utilisateur avec une connexion Wifi.

I.7.1 Travail futur

De futurs travaux à partir de ce travail de recherche peuvent inclure une plus large perspective en ce qui concerne les systèmes distribués pour graphiques 3D. D'un cté on trouve les architectures qui réalisent tout le traitement dans le client et d'un autre celles qui le font dans le serveur et transfèrent une vidéo vers le client. Comme présenté dans le premier chapitre, il existe actuellement des techniques entre ses deux cas extrmes, et en fonction des besoins des applications, une architecture appropriée peut tre sélectionnée. L'état de l'art ne couvre pas toutes les combinaisons possibles et cette formalisation peut tre développée.

La performance du terminal a permis de réaliser le rendu mais pas d'autres opérations. Des travaux futurs doivent se concentrer dans les architectures de streaming de vidéo, où, par exemple, l'utilisation du processus de rendu améliore la performance, du temps de traitement et de la qualité. Ceci peut tre fait en utilisant l'information pour que l'objet principal choisisse les paramètres de codage afin d'avoir une meilleure qualité et que le fond soit codé avec une qualité plus faible. Aussi, une optimisation du processus de codage de la vidéo peut tre réalisée en utilisant deux sorties du processus de sortie pour optimiser la détection de macro blocks pour le codage de la vidéo.

Le fait d'avoir des cas d'usage pour les différentes architectures peut tre utilisé pour concevoir une architecture qui puisse s'adapter dynamiquement d'un système distribué à un autre indépendanment de l'environnement. Il existe quelques travaux qui pointent vers cette direction mais restent limités par le nombre de possibilités offertes. Un meilleur système devrait tre capable de s'adapter à une plus large gamme d'architectures distribuées.

Chapter 1

State of The Art in Remote Computing for 3D Graphics and Games

Abstract

The first computers were very expensive and large in size. Therefore the only economic way to use them was remote computing: connect to the central computer by a light terminal. With the development of Integrated Circuits the components started to be cheaper and smaller, therefore the terminals started to support computing. Their processing power is currently enough for many applications, however when more processing power is needed, there is still the possibility of remote computing through the Internet or dedicated networks. With the arrival of mobile terminals a new iteration of remote computing is happening. By nature the mobile terminals are connected and since their processing power may not be enough for many applications, they may be helped by a remote, more powerful computer.

Therefore, the main goal of this chapter is to introduce the remote computing paradigm for multimedia applications. It will focus on a part of the multimedia application, i.e. 3D graphics applications, and present the state of the art in this field.

The first part of this chapter introduces the elements of the modern remote computing with focus on the general architectures that are used. Next it reviews the current state of the art for providing 3D graphics content in a client-server configuration. Different architectures are compared, starting from systems implementing all the operation for 3D graphics rendering on the client to distributed systems implementing the rendering on the server and streaming the video to the client. The analysis includes architecture intended for different types of terminals, from powerful computers like PCs to light terminals like mobile phones and PDAs.

The second part of this chapter presents some of the most used standardized multimedia scene formats, including VRML, X3D, SMIL, SVG, COLLADA and MPEG-4. The standards are analyzed focusing on their capabilities in terms of supported features, compression and streaming support. Then it is shown why MPEG-4 was selected as the most appropriate format and further describes the MPEG-4 standard with respect to its streaming and synchronization capabilities. Furthermore, the BIFS format is described exposing its compression and scene update capabilities.

I.1 Progress of Modern Remote Computing

What is generally considered as modern remote computing is the period after the expansion of the Internet, when different distributed applications became possible. This section analyzes the most important aspect of modern remote computing starting with a short overview of how the application architectures progressed in the last 20 years. The first applications were only providing limited services, like access to different types of data. The development of the Internet enabled to have systems that became more complex, providing more advanced features and complete applications. The further development of the Internet and also of the terminals enabled to use them for processing, hence leading to creation of mixed applications which process part of the data on the server and part on the client. Figure 1.1 presents the development of modern remote computing.

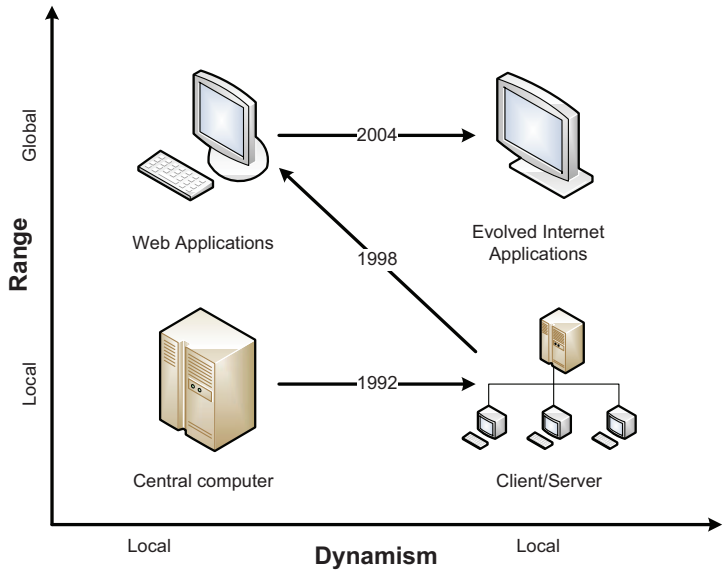


Figure 1.1: Modern Remote Computing Development

In Figure 1.1 four main stages can be identified:

- Central Computer Systems
- Client-Server Architectures
- Web Applications
- Rich Internet Applications

The next sections present in detail each of the stages.

I.1.1 Central Computer Systems

In the traditional application software, all the components are stored on one central computer, therefore accessing and processing data is performed directly. Many problems like networking and concurrent access can be avoided, thus making it easier to develop applications. However, nowadays most of the systems require exactly these features, therefore the development of traditional applications is diminishing.

I.1.2 Client-Server Architectures

Client-Server architectures are part of distributed systems where the work is separated between two parties, one that is providing services and another that is consuming them. The provider, usually named a Server, is running several server programs which provide the services. On the other side are the Clients which are using the services provided by the server, as illustrated in Figure 1.2.

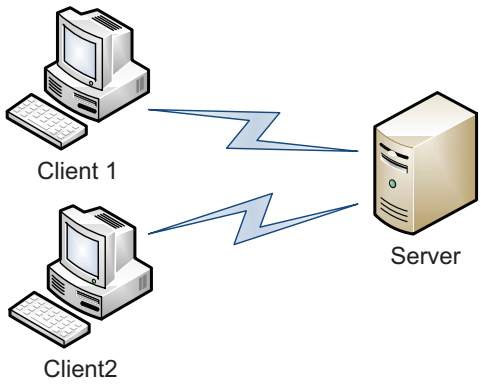


Figure 1.2: Client-Server Architecture

Figure 1.2 represents the most basic client-server architectures, named *two-tier*, one tier being the server, and the other being the client. Usually the communication between the two tiers is through a computer network, either local or through the Internet, following a specific protocol. Many applications that exist today are using the client-server model, including the main Internet protocols like HTTP, POP, SMTP and Telnet. In most of the applications the server is usually a data provider, processing only data request commands, while the processing of the data itself is left to the client. For example, in the case of HTTP servers, when the client requests a web page, the server reads the page from its storage and sends it to the client. Another example is SQL server, where the client sends the SQL command to the server where it is executed, and the result is returned.

I.1.3 Web Applications

The amount of data increases constantly, therefore the amount of processing power needed increases. Sometimes the computational power of the client is not enough to process the data, or a strong security is needed for the application, thus it is more appropriate to process the data on a server, and just return the result to the client. This means that the server side is separated in two parts: application and data, as illustrated on Figure 1.3. The applications are named *three-tier*. The processing is done in four steps: the client sends a request to the application server, which in turn requests and obtains data from the data storage server, processes it and returns the result to the client.

Web applications are a subset of the three-tier applications, where the requests are sent from and received at a standard web browser. The main advantage is exactly the web browser, since the client does not need to execute complex calculations and have big processing power. This means that the architecture can be used effectively on low power devices, like smart-phones and PDAs.

Many applications exist that are based on the web application architecture, by using the HTTP protocol for communication and the HTML standard to present the data.

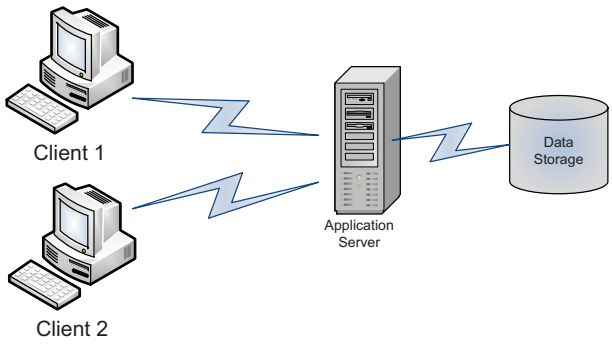


Figure 1.3: Web Application Architecture

Contrary to the standard HTML pages, which are static, in the case of web applications they are generated and served dynamically. This can be done by executing server side applications (CGI, JSP) or server side scripts (PHP, ASP). The server side applications are compiled programs which the server executes on the clients request, while on the other hand the scripts are compiled and executed when requested from the client. This allows great flexibility for creating different kinds of applications, adding features and updating.

I.1.4 Rich Internet Applications

Nowadays the trend is to have web applications which look and behave like regular desktop applications, named Rich Internet Applications (RIAs), and are usually delivered via a specific browser, a browser plug-in and independently via sandboxes or virtual machines. Therefore at the client side it is required to install a software framework that downloads, updates and executes the RIA. Some of the most popular RIA frameworks are Adobe Flex, Adobe Flash, Adobe Air, Java, JavaFX and Microsoft Silverlight. However, not all RIA technologies need a software framework, for example RIA using AJAX that exploits the already existing web browser features can deliver almost similar experience (with some limitations).

While in the past RIAs were dominantly used for on-line games, recently they started to be utilized for many other applications. Using RIAs can have many advantages:

- The complexity of implementing advanced solutions is reduced compared to traditional application software,
- Applications can be helped by servers, allowing less capable devices to run them,
- Server performance can be improved by offloading some of the work to the clients,
- The user interface is standardized between different platforms,
- Installation and maintenance is easier,
- Security may be improved by using sandboxed solutions that limit the access to the user's system storage.

I.1.5 A New Approach to Address Multimedia Applications

Complex applications such as some games require important processing power, not always available on light terminals like mobile phones and PDAs. By using the connectedness of

the devices the trend in the recent years is to have thin client-server applications, where the server makes the complex calculations and the terminal only performs the input and the visualization. So the paradigm shifts again to thin client interfaces and powerful servers. This time the thin client is a mobile device with an Internet connection that can be taken virtually anywhere. Another reason for having centralized server system is the power effectiveness of the solution. A server that is loaded most of the time is more power effective than having many PCs that are idle most of the time.

One of the most important industries behind the development of the PC is the entertainment industry, especially the computer gaming industry. There is always a direct struggle between the game producers and the computer manufacturers. The first try to produce better visual representation of the games, and the second try to create better hardware performance that can cope with the requirements of these games. The common way of playing games on PC is by buying the needed hardware. Having the latest hardware to be able to play the latest games is stressing the user's budget. And even if the user buys it, the computer will stay idle for most of the time, thus it is not cost effective. One of the solutions is a centralized system where the games are executed, and clients connect to the server to play the game [43]. One example of this kind of service is OnLive[1] which offers remote playing of the latest games. The user just needs an entry level PC with an Internet browser and an appropriate Internet connection bandwidth. The service works in a way that the game is rendered on a powerful server and an audio-video stream is sent to the client. On the other hand, the commands from the client side are sent to the server. Therefore on the client side only video and audio decoding is needed. Some set-top boxes also may fit within this service. The disadvantage of this system is that it cannot be used reliably over a wireless network. This is due to the fact that it is sensitive to network latency and jitter. While many mobile devices can now decode and render the video stream in real time, the wireless connection is not stable enough to provide constant quality. Therefore some other architecture should be used for mobile devices and this is the objective of the current work.

I.2 Distributed Architectures for 3D Graphics and Games

This section aims to present an exhaustive list of methods for accessing a game or 3D content in general within the paradigm of remote rendering.

According to the type of the data that is transmitted between the server and the client the techniques can be separated in several categories:

- Graphics commands based,
- Pixel based,
- Graphics primitives based,
- Adaptive.

The next sections will present in details each of these categories.

[1] www.onlive.com

I.2.1 Graphics commands based solution

The general idea behind the graphics commands based solution is to intercept the graphics commands sent from the application to the graphics Application Programming Interface (API) for the Operating System (OS) where the game is executed. If the OS is Microsoft Windows the API can be either Direct3D[2] or OpenGL[3]. If the OS is almost any unix kind, including Linux and MacOS, then theAPI is OpenGL.

The interception is performed by replacing the system library with one that has the same API functions. The application does not need to be changed and it will not recognize that it uses some other library. This principle is used in WireGL [39].

In the library the graphics commands are processed, compressed and sent over the network to the client. This imposes specific hardware requirements for the client. Because the actual rendering is done on the client, the graphics accelerator on the server is not used. That is an advantage because it reduces the required processing power on the server. Also it means that the display resolution on the client will not affect the processing on the server.

Let us note that some of the graphics commands require data back from the graphic card. The straightforward solution is to request the data from the client. However, because of the delay on the network, it can take some time for the data to arrive. It may not be a problem if the data is requested only once in a frame, but if more requests are made the added time easily accumulates and becomes larger, making the solution less acceptable for real time games.

A solution is proposed by Buck et al. in [18], where simulation of the state of the graphics card is performed on the server. When the game starts, the simulation is initialized with the capabilities of the graphics card on the client. After the initial request, no other request is made. When the game requests some data from the graphics card, the local simulation can return the result. The solution adds a little processing overhead on the server, but the improvement of the performance of the game is significant.

Another significant behavior can be observed related to the amount of data between the client and the server as illustrated in Figure 1.4. It can be observed that the data changes irregularly and, more important, significantly with time. This is due to the nature of the graphics commands. The largest data is transferred at the beginning of the game and when there is a scene change. The reason is that new data has to be displayed, thus new data has to be transferred to the graphics card. The data usually includes new 3D objects, textures and shaders. The impact on the bandwidth can be reduced by compressing the data and implementing caching, as demonstrated by Nave et al. in [51].

The previous limitations make this solution less appropriate for mobile devices. The main reasons are:

- The requirement to have a GPU on the client that supports the features which the game requires to render the data,

- The big changes in the required bit-rate are still not appropriate for the mobile networks.

[2] www.microsoft.com/windows/directx
[3] www.opengl.org

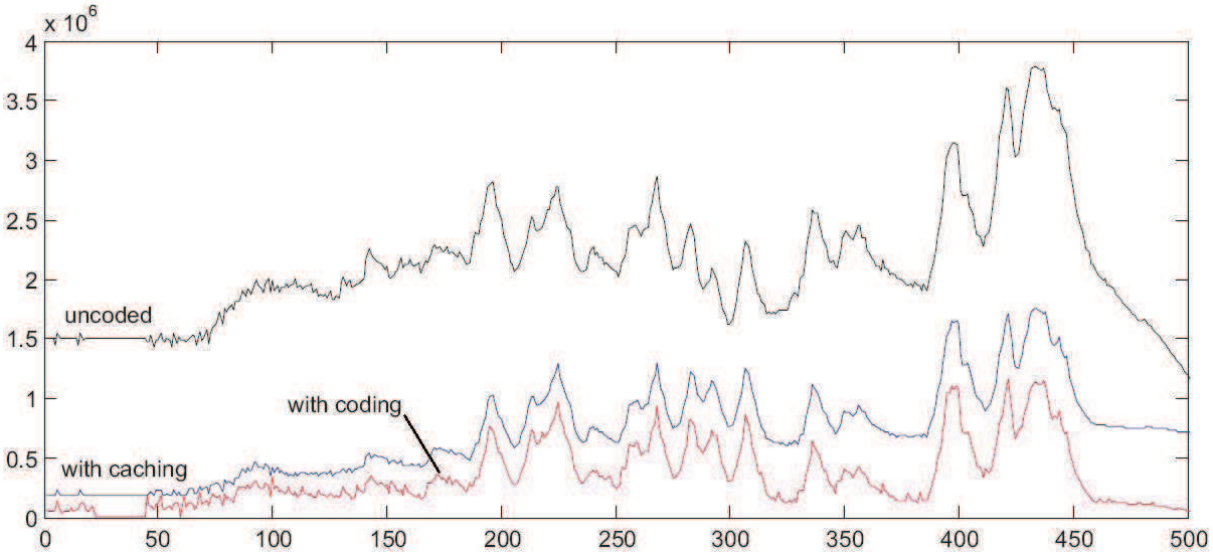


Figure 1.4: Bit-rate in bits/frame for the game Tux-Racer during game play. The upper curve denotes the uncoded bit-rate, while the lower two show the results for activated caching and additional encoding, respectively. Graph is courtesy of [51].

I.2.2 Pixels based solutions

In the pixel based category of methods the complete rendering is performed on the server. The output image is captured, and sent to the client. Depending on how the image is transferred the techniques can be grouped in three sub-categories.

I.2.2.1 Video based

The main attribute of the video based technique is the use of a video stream to transfer the rendered images. The images are captured with constant frame-rate, encoded using a video codec [23] and the result is streamed to the client. Because the complete rendering is done on the server side, it can be optimized by different techniques.

For example in the Games@Large architecture [51] the servers loads the game in a virtual machine. The games can share the graphics acceleration available on the server. In the Chromium framework [40] the server does not have to be a single computer, but it can be a cluster of computers that share the work. More complex scenes can be rendered and displayed, scenes that would normally not even load on a single computer. For instance, medical data [48], Computer-Aided Design (CAD) data, or data coming from a 3D scanner can be of several tens on gigabytes in size, making almost impossible to load and display it on a single computer and aiming at interactive applications. An example of a video streaming architecture is displayed in Figure 1.5.

On the client side, only a video player is needed. It needs to support input from streaming and to execute the required audio-video decoders. The processing power should be only enough to decode the video in real time. One advantage of using video streaming is that the bit-rate of the stream can be predicted. This allows for easier and more precise scaling of the requirements for the network traffic.

This solution is acceptable for devices connected to fast network connection. On the contrary, the mobile devices are usually connected with lower bit-rate. To accommodate the video for this network, the quality of the video needs to be scaled down, thus reducing users experience to thresholds that can easily become unacceptable.

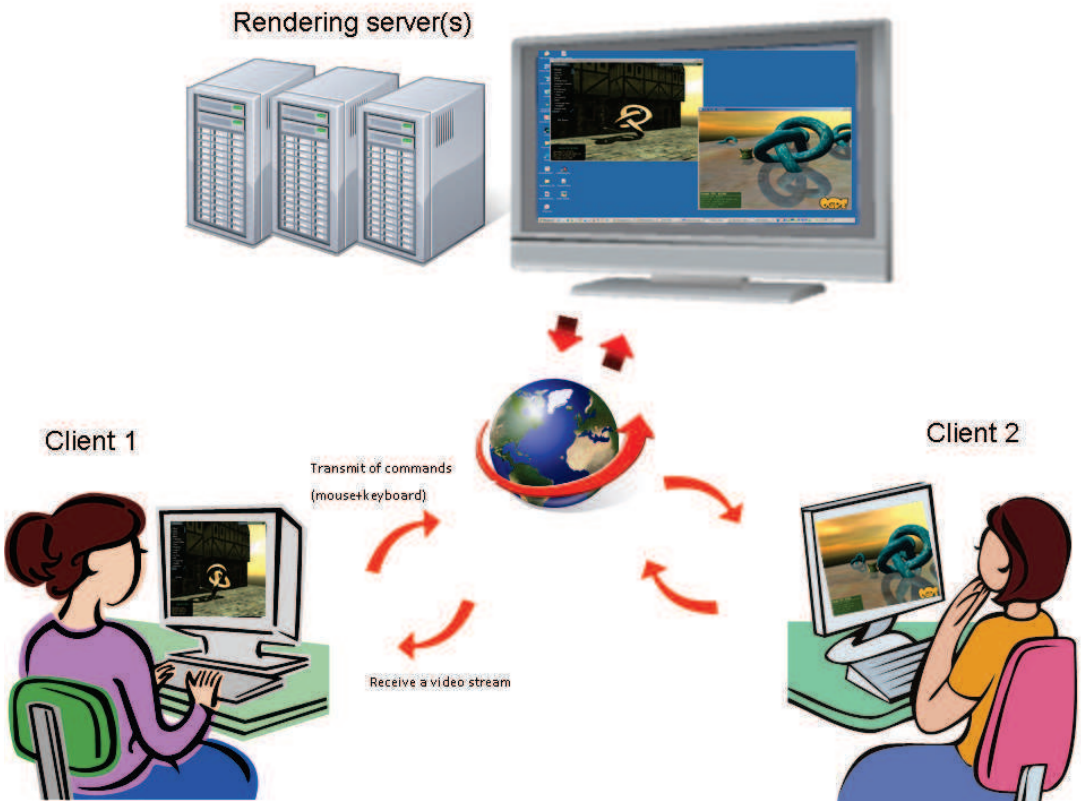


Figure 1.5: Architecture of a video based system.

I.2.2.2 Image Based

In the image based group of architectures the images are compressed and sent as standalone data [32]. In this framework the full image is transferred only when there is no movement, and a small size image while there is movement. This allows to reduce the required bandwidth, however in applications when there is movement most of the time, the quality is constantly low. The image can be captured at any time, not as in the case of video when it is captured in fixed intervals. This allows to have optimized control on the frame-rate, thus having lower frame-rate or even not sending any data when there are no changes on the displayed image, and bigger frame-rate when there are changes.

Sending constantly high quality images is appropriate for some application like mixed reality [47] and photo-realistic rendering [12], however it is not optimal for application where there is a lot of movement on the screen such as games. Then the amount of data that is sent can become very large, and the responsiveness of the system may be reduced. In some cases like virtual walkthrough applications the problem can be overcome by restricting the possible movement (allowing only predefined directions) and then predicting the next movement of the user [17]. Using this information the system can pre-render and transfer the images to the user. The images corresponding to the most probable movement are prioritized. However in most games the restriction of the movement is not an option, therefore this solution cannot be successfully applied.

An architecture of an image based system is presented in Figure 1.6.

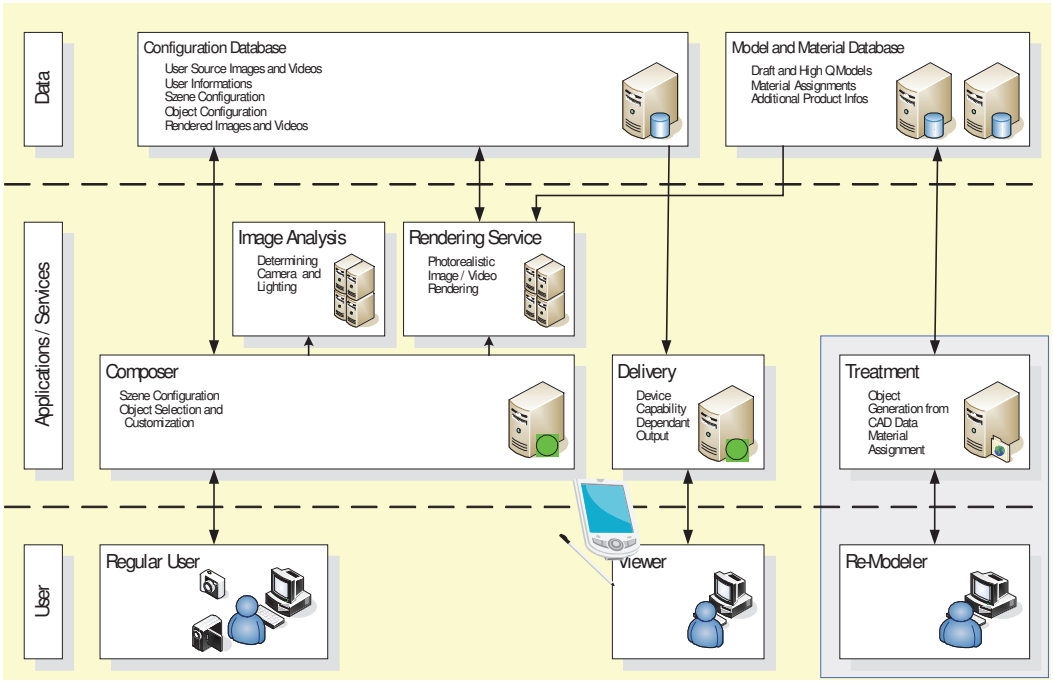


Figure 1.6: Architecture of an image based system. Image courtesy of [47].

I.2.2.3 X11 Protocol

The X11 type of architectures are primary intended for remote display of operating systems. They use the most common forms found in the OSs (windows, buttons, dialogs) and optimize the transmission protocol accordingly. One of the first remote display protocols is X11 [58]. However if the content is not in the supported group, then it is sent like raw pixels to the client.

Several attempts to optimize the X11 protocol exist. One recent optimization by Baratto et al. [15], proposed to use five basic functions instead of using all X11 functions. They also buffer the commands before sending, in order to discard commands that are overwritten by a later command, thus optimizing the network traffic. If the user plays a video on the server, then the system detects the video, and streams it to the client. However, the method does not support 3D graphics.

Support for 3D graphics was researched earlier by Stegmaier et al. in [61], where it is proposed to render the OpenGL 3D scene on the server, and then send the result as an image to the client, thus the solution becomes similar to the image based ones.

I.2.3 Graphics Primitives Based Solutions

The basic principle of the graphics primitives group of architectures is that a server is used to store the data. When the client needs to render a scene, it requests the data from the server, which in turn prepares it and sends it back. Depending on the type of the data, i.e. graphics primitives, that is sent, there are two main groups: 2D and 3D primitives.

I.2.3.1 2D Primitives

An example of the 2D Primitives based architecture is the work done by Diepstraten et al. in [28]. They introduce a new module after the rendering of the scene that converts the 3D synthetic images in 2D vectors, by extracting different kinds of feature lines and filtering them to find the 2D vectors, as illustrated in Figure 1.7.

Figure 1.7: Example of a 3D model rendered using 2D vectors. Image courtesy of [28].

Then the 2D vectors are sent to the client where they are rendered. Because of the 2D nature of the data, no complex calculations are needed, thus making it appropriate for mobile devices that do not support 3D graphics. The architecture of the system is presented in Figure 1.8.

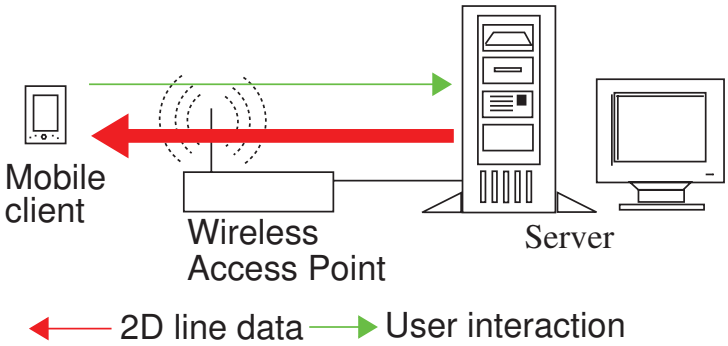


Figure 1.8: Architecture of 2D vectors based system. Image courtesy of [28].

I.2.3.2 3D Primitives

Two categories of 3D primitives can be considered: vectors and surfaces.

I.2.3.2.1 3D Vectors

In 3D vectors based technique the geometry of an object is approximated as a set of 3D vectors (Figure 1.9b), operation performed in a preprocessing phase. The vectors are sent to the client, which in turn renders them. Therefore the client needs to have support for 3D graphics. Because it renders only simple vectors, it does not need to support advanced

features like textures. Rendering lines takes less processing than rendering textured mesh, thus better rendering frame-rate may be observed.

Because the creation of the 3D vectors takes significant processing time it is done when objects are loaded, not when they are requested by the client. The processing is done by rendering each object with the associated texture (Figure 1.9a), and then extracting the vectors. The method allows having more details on the objects than in the case where only the contour of the objects is extracted like in the previous mentioned 2D vectors technique.

(a) Using original images (b) Line-based rendering

Figure 1.9: Comparison between rendering using textures and line rendering. Images courtesy of [57].

An architecture based on a 3D vector approach was proposed by Quillet et al. in [57] where they use it for displaying 3D city model for navigation purposes. They tested the software using a mobile device and conclude that this type of rendering is appropriate for recognizing building facades. The architecture representing this system is displayed in Figure 1.10.

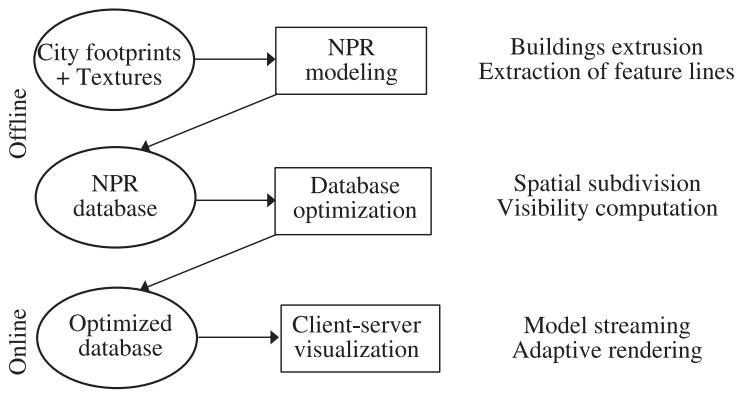


Figure 1.10: Architecture of 3D vectors based system. Image courtesy of [57].

I.2.3.2.2 Surfaces

In the surfaces group of techniques complete 3D graphics objects (i.e. surfaces) are transferred. At the client side 3D rendering has to be supported. Because portable devices are taken into account, the server has to be aware of the capability of the client device to be able to adjust the content accordingly. The techniques can be separated in two main groups. In the first group only one 3D object is transferred and optimization is done per object. In the second multiple 3D objects are transferred and optimization is performed for the entire scene as a sub-scene.

Single Object

Most of the challenges of transferring single objects are connected to the unreliability of the network. The problem arises when the objects need to be transferred with short delays. The TCP protocol is reliable, but introduces a significant delay over lossy networks. On the other hand, the UDP protocol has a shorter delay, but it is unreliable. Because of the importance of the delay, most of the interactive 3D applications use the UDP protocol.

The unreliability of the protocol can be partially overcome by adapting the data with usage of error resilience. Depending of the way how the data is adapted, the algorithms can be separated in two categories: segmentation and progressive.

The segmentation algorithms [14, 67] separate the mesh into more parts (segments) that are transferred to the client separately. If a part of the mesh is lost, one can still see the other parts. The lost part can be interpolated, but this requires additional processing from the client. Different techniques exist that optimize the segmentation of the mesh for improving the visual results.

The progressive algorithms remove vertices from the mesh, thus simplifying them as it was proposed by Al-Regib et al. in [11, 10]. The details about the removed vertices are stored and grouped into one data structure, being possible to restore the details to the mesh at the receiver side. Removal of vertices is done in several steps, until a basic shape with minimum number of vertices is acquired. These steps are called Levels of Detail (LODs). Since restoring vertices depends of the previous LOD, if that LOD is lost, it is not possible to restore the consequent LODs.

The biggest problem can arise when the basic mesh is lost. It means that the next LOD that are received cannot be reconstructed and no mesh will be displayed in the scene. Therefore the base mesh is usually sent by more secure means, either by using TCP or a secure UDP transfer.

Multiple Objects

The challenges in transferring multiple objects are mostly connected with the order in which the different objects are sent to the client. Usually, objects in a scene are organized in some kind of graph, called scene-graph. The scene-graph contains the position of each object in the scene, how different objects are grouped together and their connections. To save memory, it is also possible to have multiple occurrences of the same object in different parts of the scene. The object is not copied, but an instance of the object is created in that branch of the graph. However, in most of the cases, the scene-graph structure is simple and flat, including at most one level, and therefore it is not considered for compression.

One of the most significant types of architecture is the one intended for virtual cities maps where a large amount of data should be considered. Therefore not all of the data is transferred at once but in a progressive manner. The data is chosen by the location

of the virtual camera. Only buildings that are near are rendered in full details. This kind of architecture was proposed by Nurminen in [53] and later improved in [54]. The buildings are grouped by their location in the 3D space, however only the groups that are inside the viewing frustum and closest to the camera are transferred to the client. Inside each group, the buildings are again compared with the frustum, and ordered by the distance to the camera. According to the distance, different LODs are used, and only the needed LODs are loaded. The most distant buildings are not textured at all, though uniform color is used instead. The special buildings, like landmarks and monuments, are processed separately. If they are far from the camera, only a billboard is used. Billboard is a rectangle mesh with one texture that is always facing the camera, thus only giving an impression that the object is 3D. An architecture of a such a system is presented in Figure 1.11.

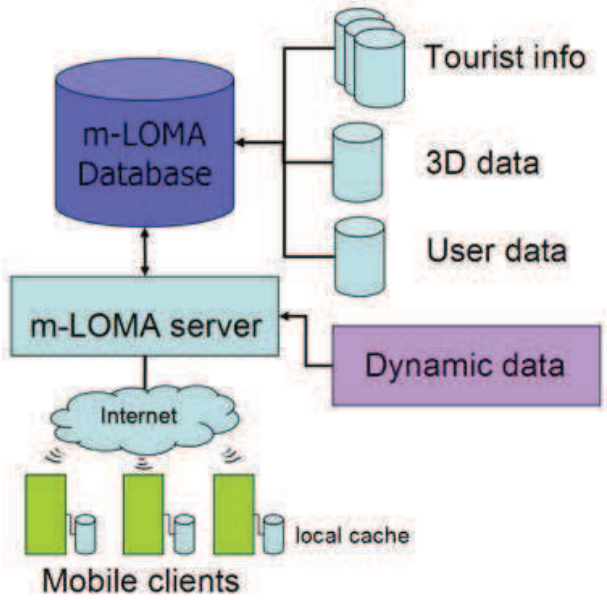


Figure 1.11: Architecture of multiple 3D objects based system. Image courtesy of [53].

The architecture proposed by Coors in [27] uses more diverse measures to compute if a building should be sent to the client. The buildings are ranked by calculating an importance map. The system, based on an Oracle database, creates a map for each user based on his queries. The system learns what the user searched in the past, and it uses this knowledge for ranking the results. The ranking also takes into account different attributes that are specific for a building and technical resources like the hardware specification of the user device and the available bandwidth.

I.2.4 Adaptive methods

The adaptive methods use a combination between two or more of the previous mentioned methods. The methods can be grouped by several criteria such as motion, distance, importance, terminal power.

I.2.4.1 Amount of motion on the screen

A method proposed by De Winter et al. [66] detects how often and how much the image on the screen changes. If the changes are small, then an image based method is used.

When the frequency of motion increases, the system switches to a video based method. The advantage of the adaptation is that when the screen changes are infrequent, the bandwidth usage is small. When the amount of motion increases, for example in the case when movie or game is played, the system stays responsive without large increase in the needed bandwidth.

I.2.4.2 Amount of camera motion

The architecture for this method uses two LODs models for the objects in the scene. The model with lower LOD is used by the client and the higher LOD model is used by the server, as illustrated in Figure 1.12. When the user moves around the scene, then only the lower LOD model is displayed. When the movement of the user stops, the client sends the camera parameters to the server. Then the server renders a better quality image using the high LOD model and sends it back to the client. This system, proposed by Koller et al. in [46], is not intended for application where the client movement is almost continuous, such as in games.

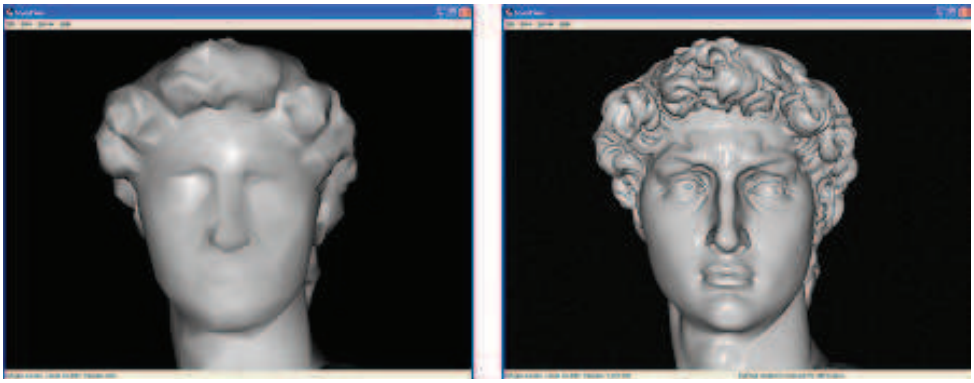
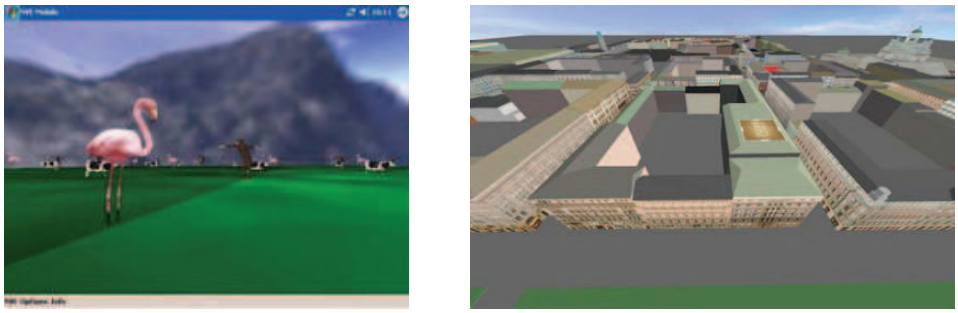


Figure 1.12: Client-side low resolution (left) and server-side high resolution (right) model renderings. Images courtesy of [46].

I.2.4.3 Distance from the camera

In the case when there are many objects in a scene, usually the ones that are closest to the camera have more visual impact that the rest. Therefore by sorting the objects with respect to the distance from the camera, some priority can be assigned. Depending of the capabilities of the target device, the closest objects are sent with full detail, the next objects are sent with less detail, and the furthest objects are either not sent at all, or simplified to the lowest possible detail.

Jehaes et al. in [41] proposed that the objects closest to the camera are rendered on the target device, and all other objects are rendered on the server (Figure 1.13a). For the later only a billboard representation is sent to the client. This helps to save bandwidth (billboard objects take less space than full 3D models), increase the number of frames per second (FPS) at which the scene is displayed (billboard objects are rendered faster than 3D models), and present the scene with acceptable image quality (although the quality of the billboard is lower than that of the 3D object, because it is far from the camera, it will not be very noticeable). Another example is the method proposed by Nurminen in [53] and [54], where the buildings that are far from the camera are displayed with fewer details and using flat colors instead of textures (Figure 1.13b).



(a) Billboard based (b) Color based

Figure 1.13: View dependent rendering of 3D objects. Images courtesy of [41] and [53].

I.2.4.4 Importance to the user

In city a navigation scenario, users often search for a specific location, apart from just browsing the city. Hence, a criterion is how much the user is interested in a specific object (i.e. location) in addition to the users location. By using this information, a research was done by Coors in [27] where they use a database query to retrieve the objects and it is illustrated in Figure 1.14.

Figure 1.14: Objects that are important to the user rendered in more detail. Image courtesy of [27].

For example, if the user searches for available hotels in some area, the hotels found are displayed with maximum available detail, the surrounding buildings with less details, and the rest of the scene is only 2D.

I.2.4.5 Processing power of the users terminal

Sometimes there is a huge amount of data available that needs to be processed and rendered. Usually one terminal is not capable of rendering all of the data, hence a simplification is needed. This was researched by Grimstead in [36], where he proposes that the data is sliced in smaller parts and only the quantity that can be rendered on the target terminal is sent. If the target terminal is not powerful enough to render even a part of the data, video streaming is used.

I.2.5 Analysis of the different architectures and conclusions

Figure 1.15 presents the different architectures reviewed in the previous sections. The methods are ordered from left to right first by their dependence of a server. Let us note that three categories may be observed: (1) game logic and data on the client, (2) game logic on the client and data on the server, and (3) game logic and data on the server.

On the far left side are the techniques that have everything on the client, whereas on the far right side are the techniques that have everything on the server, and only display the end image on the client. The techniques are separated in two main groups: the first one, containing the techniques that execute the application logic on the client and the second one, containing the techniques that execute the application logic on the server. The first group is separated in two main parts: one for techniques that store the data on the client and the second part for techniques that store the data on a server.

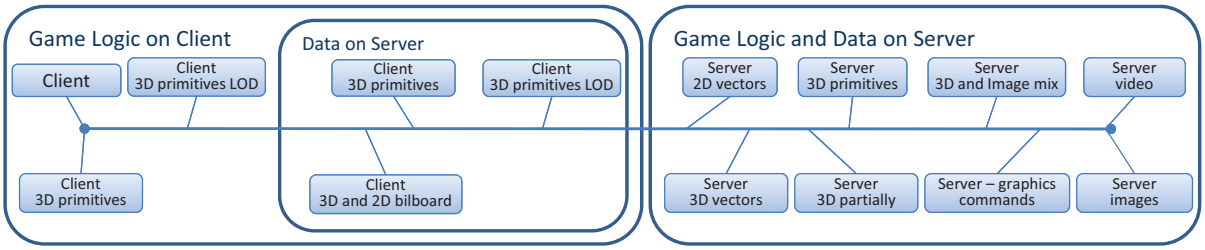


Figure 1.15: Techniques for displaying 3D graphics

By analyzing Figure 1.15 some conclusions can be withdrawn. According to the state of the art research done by Capin et al. in [20], there are numerous solutions that can optimize graphics rendering on mobile devices, however they are still limited by the devices' processing power. For example, developing a visually beautiful chess game would require reducing of the computer chess player capabilities in order to support the more beautiful graphics. Capin et al. conclude that remote rendering can be a viable solution, and a solution where the processing is balanced between on-device and remote rendering represents an interesting research direction. Therefore this thesis continues in this direction and focuses on the architectures that have the game logic on the server and rendering on the client.

As it was presented previously, they can be separated in six main categories:

1. Graphics commands
2. 2D pixels
3. 2D primitives
4. 3D vectors
5. Single 3D object
6. Multiple 3D objects

For these specific architectures additional observations can be made:

The Graphics commands based architecture can be dismissed immediately because of the hardware incompatibility between mobile devices and PCs.

2D pixels The 2D pixels based architectures have two main disadvantages that are closely connected: required network bandwidth and visual appearance. When streaming video or images, the required network bandwidth is relatively high. Reduction of the bandwidth can be achieved by increasing the compression ratio, however, this implies that the quality of the video or images will be reduced. Furthermore, if the video stream saturates the channel, the latency between the user commands and the respective response from the server will also increase, hence the user experience will be reduced. Therefore a balance has to be sought between these two components. In the case of mobile networks, the available bandwidth is not enough to achieve good video quality because the synthetic content is more sensitive to compression artifacts (e.g. blocking) than natural content. The processing power on the terminal side has to be enough to support real-time decoding of the input video stream or images. Therefore it can be concluded that these architecture do not satisfy all requirements simultaneously.

2D primitives The 2D primitives based architectures require relatively low bandwidth that can be supported by the mobile networks, however the visual quality of the rendered image is significantly reduced. The limitation of this approach is that the contours have to be recalculated and resent each time the scene view changes. When dealing with simple scene objects, promising results may be observed. When the number of objects increases the detection of contours can take more of time on the server, thus making it less appropriate for real-time applications. A second problem is the quality of rendered image on the client side. Because of using only contours, there is a significant loss of visual quality. For some applications this may be satisfactory, but for richer media applications like games, such artifacts are opposite to the users' expectations. Some improvement in the visual quality can be made by adding color description for the lines, however it increases the size of the transmitted data. The capabilities on the client side have to be enough to support rendering of 2D primitives, which is the case for most mobile devices.

3D vectors An advantage of this architecture with respect to the 2D primitives one is that the 3D vectors for one object can be sent only once, and the user can navigate into the scene without additional data, meaning that the bandwidth requirements are lowered. Because of the line rendering, far objects can be cluttered with lines, so LOD adaptation based on the distance of the objects from the virtual camera may be implemented. The principle consists in removing the least important (shortest) lines as the distance increases. Because only 3D vectors are rendered, the system still has image quality drawbacks as for the 2D primitives based. However, the processing requirements at the client side include support for rendering simple 3D vectors, an operation that can be handled with ease by the most devices.

Single 3D object The single 3D object architectures handle the transferring of a single 3D object from the server to the user, however they do not handle the relationship between different objects. Therefore they can not be used directly for representing complex scenes such as the ones in games. Furthermore, these architecture do not include any command from the client side, apart from the request for an object. Because the full 3D object is transferred, the rendering quality at the client side is maximal. Furthermore, the client has to support rendering of a full 3D object including texture, however this is easily handled by current mobile devices.

Multiple 3D objects The multiple 3D objects architectures are mainly used for city navigation applications, hence the only operation that the server performs is to order the objects with relation to the position and the orientation of the camera and then send them in that order. This means that the server can send data that will not overwhelm the capabilities of the user terminal, however it also means that there is no support for complex manipulation of the scene-graph like semantically grouping of objects that is needed for games. Furthermore, the only communication sent from the user terminal to the server is the data with relation to the camera, however this is not sufficient for game applications where the camera is not controlled directly by the user, but indirectly trough the game logic. This is important because it means that the user should send key actions to the server with latency as low as possible. Because the objects are sent only once, it can be concluded that the required bandwidth will be within the limits of the mobile networks.

The following statements summarize the analysis:

- The techniques that use image based or graphics commands based transfer are not suited for mobile networks because of their bandwidth requirements,
- The techniques that transfer 3D or 2D graphics primitives are better, but what they lack is an appropriate control on the data, in the sense that no scene-graph is used for organizing the data,
- The techniques that convert the data in lines (2D or 3D) are not visually satisfactory.

Therefore none of the architectures satisfied simultaneously all requirements, hence an alternative architecture that can be applied for games for mobile devices is needed. It was observed that the scene-graph organization has a major importance for the game applications. Therefore the next section will present the state of the art of the different scene-graph formats. Furthermore, the most appropriate scene-graph format will be selected and described in more detail.

I.3 Multimedia Scene Description Languages

The term *Multimedia* refers to media that can take multiple forms for representing and processing information (i.e. text, audio, graphical elements, video, interactivity), with the objective to inform or entertain the user or observer.

To be able to have coexistence of the different types of media, it is necessary to manage their presentation, both temporally and spatially. Videos and images are displayed as 2D arrays of colored pixels, sound is sent to the speakers, and 3D objects are placed into a virtual scene and can be seen through the viewpoint of a virtual camera.

Managing the different media can be done by combining them into one entity, named Multimedia Scene. It defines the logical, temporal and spatial relations between different media in a scene, organized like a collection of nodes connected as graph or a tree. A node can have multiple children, and in the case of the graph it can also have multiple parents. An operation executed on the parent propagates to the children. For example, at each node a geometrical transformation matrix can be associated, therefore allowing control of the position of all children of that node at once. Historically, the scene graph concepts are inherited from the techniques enabling to optimize 3D graphics rendering. In

order not to process invisible objects (i.e. objects being outside the view frustum) a spatial organization of relations between them is created. Additionally, common properties such as textures or lighting conditions may be used to group together different objects; consequently, loading and unloading operations per object are now performed per group of objects.

A Multimedia Scene Description is usually referred to as scene-graph. Through the history different scene-graph representations were developed. Because of the lack of a standard that can satisfy the needs of any scene, the producers of multimedia content and multimedia viewers developed proprietary scene-graph representation. Nowadays this is not practical due to the development of the Internet and the requirement of the interoperability of the media content. If different multimedia platforms need to access the same content, one should create a converter from the content format to the specific application format. This can be tedious considering the number of proprietary formats.

I.3.1 Review of scene-graph formats

In the following sections different standards for scene description will be analyzed showing their advantages and disadvantages.

I.3.1.1 VRML and X3D

Virtual Reality Modeling Language (VRML) [5] is one of the first open standards related to 3D graphics and its second version (the firtst to be an ISO standard) was released in 1997. The main target of VRML was the publishing of 3D models and worlds over Internet, by implementing similar functionalities as HTML does for text. Therefore VRML is a textual format based on a human readable description, thus making the integration into different content creation tools easier.

VRML does not have support for 2D graphics, having only 3D graphics primitives that can be presented. There are four ways of presenting them:

- Basic 3D primitives (Box, Cone, Cylinder and Sphere),
- Points (PointSet),
- Triangle meshes (IndexedFaceSet, IndexedLineSet, ElevationGrid, Extrusion and GeoElevationGrid),
- NURBSs (NurbsCurve, NurbsSurface and TrimmedSurface).

VRML allows creating files by using an inclusion mechanism that uses hyperlinks to refer to an external file, re-utilizing data by referencing mechanism (DEF, USE), re-utilizing complex content by macro definitions (PROTO, EXTERNPROTO). Another important feature is the support for user interactivity actions by using special nodes names Sensors (e.g. TouchSensor, TimeSensor and ProximitySensor). The user interactivity is closely connected to the scripts supports by using the Script node. The scripts are typically used to:

- Signify a change or user action;
- Receive events from other nodes;

- Contain a program module that performs some computation;
- Effect change somewhere else in the scene by sending events.

Therefore it can be used to create not only scenes, but also applications. However because of the interpreted nature of the scripts, they are not well suited for use for complex computations on constrained devices.

One disadvantage of VRML is the lack of compression support. Streaming of graphical content is also not allowed by VRML, making difficult to update the scene graph in real time applications.

eXtensible 3D (X3D) [6] is the successor of VRML proposed by Web3D and published in 2005 by ISO. It extends VRML by presenting the data using an XML schema and adds new nodes. X3D is still being developed, integrating more advanced features that previously mentioned standards, for example support for shaders, geospatial component, particle systems, physics and different input devices. The compression of X3D became possible with the introduction of X3D binary, based on compression methods that compress data based on its type, therefore utilizing its properties. X3D supports streaming of only video and audio, but not 3D content, making it inappropriate for server types of 3D multimedia applications.

Despite some advantages over VRML, X3D is not accepted widely by the industry, limiting its usage mainly to research.

I.3.1.2 SMIL

Synchronized Multimedia Integration Language (SMIL) [19] is another standard for scene-graph description released in 1998 by WRC. The intention was to allow integration and synchronization of a set of independent multimedia objects into one presentation. The latest version 3.0 was released in 2008.

SMIL is an XML markup language that defines timing, layout, animations, and visual transitions for presenting different media items that include text, images, video and audio. The standard supports only reference to 2D graphics, and does not support 3D graphics. For example it is used to represent animations in SVG files. Objects can be displayed anywhere on the screen grouped organized in layouts and regions. Additionally for each object timing information can be added, e.g. start time when the object should be displayed, duration for how long the object is displayed.

Since SMIL uses hyperlinks for referencing the content, it is also used as a playlist file. A hyperlink can be associated for each object, therefore SMIL can also be used for creating interactive applications like the one used for the HD DVD format.

As VRML, SMIL does not support compression of its format, although the content that is referenced is usually compressed.

I.3.1.3 SVG

Scalable Vector Graphics (SVG) [64] is an XML based language created by W3C and released in 2001, that is used to describe 2D graphics and graphical applications. Unlike bitmap formats that use pixels for representing data, SVG describes the scene using 2D graphics primitives like polygons, ellipses and Bzier curves, a format also known as vector graphics. Since objects are defined parametrically, the scene can be viewed on any resolution with the best possible quality. Objects can be displayed by using only their

outline, however they can also be filled with one solid color or a gradient of colors. An example of rendered SVG image is displayed in Figure 1.16.

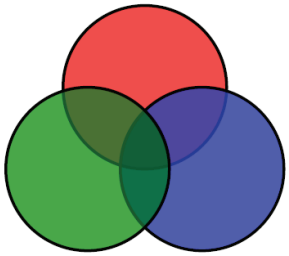


Figure 1.16: SVG Sample Image

On the other hand SVG supports scripting by using ECMAScript [9], hence all objects can be accessed and manipulated in similar way to HTML. Furthermore, scripts can be used to animate objects and respond to user or document generated events. All of these features allow for creating richer scenes and applications.

In 2003 W3C released SVG Tiny and SVG Basic [65], two simplified versions of SVG intended for use on mobile devices. SVG Tiny is intended for mobile phones, while SVG Basic is intended for PDAs and smart-phones. These specifications removed many of the features of SVG, making it less demanding in terms of memory and processing. One of the removed features was the scripting support. Therefore SVG on mobile devices is used primary for displaying static or animated images without interactivity.

SVG has the same disadvantage as the previous standards, the nonsupport for streaming of the content. However some compression support was added with the introductions of the SVGZ format which is basically a SVG file compressed by gzip. Although the compression may be satisfactory, it only compresses the file like a simple textual file ignoring any properties of the content itself (XML structure, 2D primitives etc.).

I.3.1.4 COLLADA

COLLAborative Design Activity (COLLADA) [44] is designed to be an intermediate format between Digital Content Creation (DCC) tools. Sony Computer Entertainment released the initial version in 2004, however since 2006 it is released through Khronos. Unlike the previous standards, COLLADA is widely accepted by the industry, being supported by number of DCC tools and game engines.

Given that it is mainly used as an intermediate format, it supports many 2D and 3D graphical features [13], even some that are not used in games. The format can be used to specify how the data is displayed on different terminals through the specification of different profiles, therefore allowing the artist to tweak the data specifically. Additional resources like images for textures are referenced by using hyperlinks.

The format is based on XML. Besides the strong support for 2D and 3D graphical primitives, other type of media like video or sound and are not supported. This is a big disadvantage since it means that COLLADA by itself cannot be used for storing data needed for more complex multimedia applications and scenes. An additional disadvantage is that there is no support for user interactivity or scripts, therefore limiting its usage only for storing data. Furthermore there is no support for compression and no support for streaming of data.

I.3.1.5 MPEG-4 Scene Description Languages

The MPEG-4 standard defines three manners of describing scenes: BIFS, XMT and LASeR. Each one is used for a specific targeted application. The next subsections explain each representation.

I.3.1.5.1 BIFS

MPEG-4 BInary Format for Scene (BIFS) [4] is specified in chapter 8 of the standard ISO/IEC 14496-1 released in 1999 and then revisited as a Part of the standard ISO/IEC 14496-11 in 2005. BIFS is a binary compressed format which is based on the VRML specification, extending it with additional functionalities including streaming, synchronization and protection.

Like the VRML standard, BIFS can represent 3D scenes and object, however with the addition of the 2D nodes, BIFS can also manage 2D scenes or mixed 2D/3D scenes. The last functionality is made possible by the mechanism of layers. On the other hand BIFS allows changing scene dynamically by making updates: scene elements can be added, deleted, changed or replaced. This process changes the static representation of a scene into a media stream that can be sent over a network and synchronized with other streams like video or sound. Two mechanisms exist that allow the implementation of updates: BIFS-Command and BIFS-Anim.

MPEG-4 BIFS functions with the MPEG-4 Object Descriptor Framework (ODF), which allows unifying of the management of different types of media. For example, a JPEG image and a video are accessed in a same way by using an OD for the image or the video, respectively. An OD holds information about synchronization, encoding and description for the media referenced by the object.

I.3.1.5.2 XMT

EXtended MPEG-4 Textual format (XMT) [45] is a standardized XML description of the BIFS scene. It has two levels of abstraction. The first one, named XMT-A, is in a direct correspondence with the binary BIFS, therefore it is very close to X3D. The second one, named XMT-O, is an abstraction on a higher level, which has SMIL elements and which allows for deployment of fast applications with lower complexity.

XMT can be compressed by applying a generic XML compression tool standardized by MPEG and called BiM, therefore allowing deployment through networks with lower capacity. However, XMT has only limited support for compression of 3D graphics streams.

I.3.1.5.3 LASeR

Lightweight Application Scene Representation (LASeR) [29] is designed for usage on low performance terminals such as mobile phones by reducing the number of supported features, i.e. nodes. One of the features removed is the support for 3D graphics, therefore making LASeR the equivalent of SVG. Therefore it can be said that LASeR combines the SVG elements with compression and streaming.

I.3.1.6 Analysis

By looking at the previously mentioned standards it can be concluded that all of them are targeted at different applications. Choosing which standard to use depends of its support

for these main requirements:

- 3D graphics: to be able to represent 3D graphics as well as animation;
- Compression: to reduce the size of the transmitted data;
- Streamability: to enable players to start using the content before having downloaded it entirely.

Table 1.1 summarizes the capabilities of all standards presented previously. It can be observed that SMIL and SVG do not support 3D graphics, nor streaming or compression. VRML and X3D have support for 3D graphics, however there is no support for streaming. Furthermore they lack support for compression. Although COLLADA is very good as an interchangeable format, it lacks any support for streaming and compression. As it can be observed from Table 1.1, only the MPEG-4 standard satisfies all requirements, being the only one able to deal with 3D graphics, its streaming, compression and offering the possibility to update the scene during run-time. Therefore in the next section a description of the MPEG-4 standard concerning scenes is presented.

Table 1.1: Comparison of the Multimedia-Scene description Standards

Supported features:	VRML	X3D	SMIL	SVG	COLLADA	MPEG4	
						BIFS	LASeR
Primitives:							
Text	X	X	X	X		X	X
2D Graphics			X	X		X	X
3D Graphics	X	X			X	X	
Audio	X	X	X			X	X
Video	X	X	X			X	X
Animation	X	X	X	X	X	X	X
Streaming:							
2D Graphics						X	X
3D Graphics						X	
Audio	X	X	X			X	X
Video	X	X	X			X	X
Animation						X	X
Synchronization	X	X	X			X	X
Special:							
Compression						X	X
Interactivity	X	X				X	X
Client Events						X	X

I.3.2 Description of the MPEG-4 standard

I.3.2.1 Introduction

Bringing together the socio-economic challenges in the world of multimedia, the MPEG (Moving Picture Experts Group) of ISO has promoted the audio compression standards

MPEG-1 and MPEG-2, with a proved success in television, CD-ROMs and DVDs, without forgetting the famous "mp3".

Meanwhile the MPEG-4 standard [7] marks a decisive leap in the evolution of audio and video coding technologies. By adopting the paradigm of a new generation of compression tools, MPEG-4 implements object-oriented representations (natural and arbitrary or synthetic shapes) and the principles of selective coding, while is supporting diverse range of media and interactive applications.

Going far beyond the audio and video objects, MPEG-4 version 2 also specifies a rich set of technologies dedicated to the management of 2D or 3D scenes, in the definition of graphical primitives and their compression (applicable for the most demanding storage capacity such as 3D meshes and animation of virtual characters), as well as user interactivity.

I.3.2.2 Usage Domains

The primary objective of the MPEG-4 standard is being the successor of the MPEG-1 audio - video compression standards and the MPEG-2 standard for digital television. However when developing the standard, many applications and features have been added and continue to be added, thus surpass the initial goal. Therefore MPEG-4 grows into a real revolution both in its concept and the number of applications it touches.

MPEG-4 is the merger of three worlds: computers, telecommunications and television. It is the result of an international effort involving hundreds of engineers and researchers around the world and from diverse backgrounds: universities, research centers, major computer companies (IBM, Microsoft, Sun...), telecommunications (AT&T, France Telecom...) and other major industrial groups (Philips, Sony...).

The first version of MPEG-4 was finalized in October 1998 and since then new versions have been integrated, providing both quality improvements in terms of compression and new features. The development of the standard is motivated by the success of digital television, interactive graphics, entertainment systems and multimedia (WWW). Nowadays the most popular means of expression on the Internet is the video content created by the users, and uploaded on web services like YouTube, Vimeo, and Dailymotion, having some of them already use MPEG-4 video and audio compression to improve their user experience. However in the next years it is expected to have another revolution, this time in the field of on-line 3D graphics. Even today there are on-line services that allow users to contribute and share 3D content (e.g. Google warehouse), consume 3D content (e.g. GoogleEarth), although they are still not as expanded like video.

Interactive digital TV, mobile telephony, Internet, video games are now the major applications of MPEG-4 that must adapt seamlessly to the user, the constraints of available resources both in terms of transmission and reception requirements at the client terminal. Therefore MPEG-4 provides technologies not only for compression, but also for adaptation of the content and the tools themselves dependent on the resources at the terminal. In general it can be said that for each type of content, there are tools available that can optimize it for the targeted terminal. For example one can choose slightly less effective 3D compression to get lover complexity of the decoder because of processing constrains on the user terminal.

As a result, the MPEG-4 standard is also looking into the future and preparing for it, its purpose being to ensure a technological standardization at all levels: production, distribution and streaming for all types of media. Therefore it provides a set of technologies satisfying the needs of authors, vendors and end users.

- For authors: MPEG-4 enables the production of reusable resources. It allows great flexibility, allowing the mix of digital television, animated graphics and web pages. In addition, they can protect their creations.
- For ISPs: MPEG-4 offers transparent information which allows easily adapting to the user request (e.g. adjustments to the language of the user) and controlling transfer (management of data loss).
- For users: MPEG-4 has many possible applications and features that can be accessed from a simple terminal. The wide range of applications covered by the contributions of an MPEG-4 solution include:
  - Real-time communication (videophone),
  - Video monitoring,
  - Multimedia mobile devices (mini laptop acting as a telephone, fax, calendar, ... via GSM or satellite),
  - The storage and retrieval of information based on the content (in connection with MPEG-7),
  - Video playback on the Internet / Intranet without having to download the whole source (streaming),
  - Visualization of a same scene simultaneously at several terminals (conference call)
  - Transmission of any data type,
  - Post-production (film and television),
  - The next generation of DVDs, Blu-ray discs, etc,
  - Applications of character animation synthesis and 3D worlds (virtual meetings, on-line games, ...) based on Part 16 (AFX) [8] of the standard,
  - The prioritizing and management of audio objects in a scene.

I.3.2.3 MPEG-4 File and Stream Organization

As it was discussed previously, the MPEG-4 standard is organized in a way that it provides many features useful for different applications. Therefore in this section its organization will be presented.

I.3.2.3.1 Fundamental concepts

A scene may be composed of many individual audio or visual entities, usually named audio-visual objects. In the case of MPEG-4 they can be either natural (i.e. recorded) or synthetic generated by the mechanisms of BIFS (illustrated in Figure 1.17). However BIFS is not just a scene description language, it integrates both natural and synthetic content into one presentation space. Therefore some objects may be described fully in the scene description, while other are just referenced, having their data in other separate channels. This coincides with the general paradigm of MPEG-4 that all information is conveyed in a streaming manner.

In the MPEG-4 terms, channels are named elementary streams (ESs), containing each a fully or partialy encoded representation of a single audio or video object, scene description, or control information. ESs, in turn, are identified and characterized by object

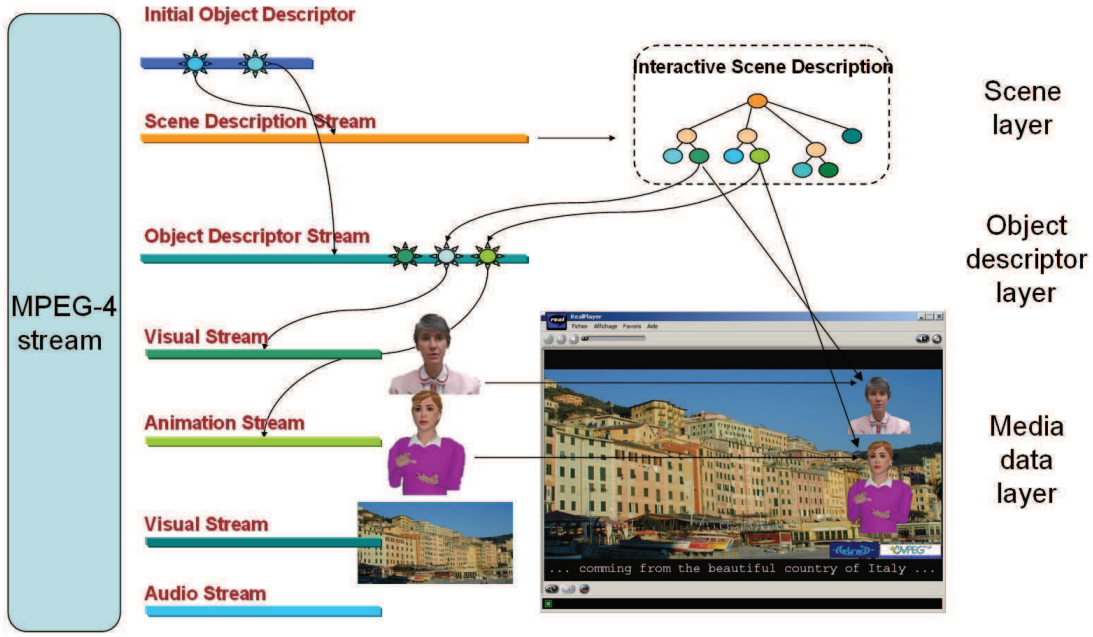


Figure 1.17: MPEG-4 terminal data flow

descriptors (OSs) which are also stored as ESs and contain the format of the data (e.g., Bone Based Animation stream), a configuration specific for the decoder, as well as indication of the resources needed for decoding. ODs can also reference to multiple streams for single audio-visual objects, enabling their scalable and adaptable representation. Therefore the most important feature of an OD is that it contains all necessary information to identify the location of the data for a stream either in terms of an actual transport channel or a URL. This allows having an indirect approach and separating the internal handling of the streams from the network and/or protocol-specific addressing.

I.3.2.3.2 Object Descriptors

Besides being considered as ESs, the scene description and the stream description are strictly separated, meaning that the scene description does not contain any information about the streams that it uses, whereas the stream description does not contain information of how it is used in the scene. This mechanism allows having room for some other meta-information, as well as facilitating editing and general manipulation of the MPEG-4 content. The stream descriptor follows the same concept as the other data, having the form of an elementary stream. The link between the two descriptions is a numeric OD identifier (OD_ID) that the scene description uses to point to object descriptors, which in turn provides the necessary information for assembling the ES data required to decode and reconstruct the object at hand. The OD object integrates other descriptors including ES descriptors which describe the individual streams that are associated with it. Additional auxiliary descriptors include textual description of the content embedded into Object Content Information (OCI) descriptor and Intellectual Property Management and Protection (IPMP) descriptor as illustrated in Figure 1.18.

The integration of a stream into the scene is a two-step process: first the scene uses the OD_ID to reference the OD; and a second step, the OD uses the ES identifier (ES_ID) stored in the ES Descriptor to reference the stream itself. This organization has two main advantages: first, more streams can be grouped together and second, each of these groups

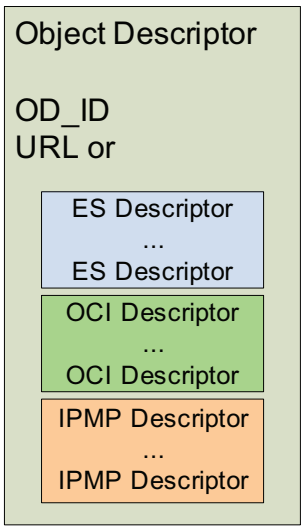


Figure 1.18: Object Descriptor contents

can be accessed by different scene elements. The grouping of the streams allows having an adaptability of the scene to the user preferences, for example the user can choose from many possible languages for an audio stream, or for example the user can receive a stream with different quality based on the available network bandwidth. It should be noticed that grouping of streams in one OD is to be done only for streams that share the same type. Therefore different stream types, as well as streams that are not related, will need different ODs.

As illustrated in Figure 1.18, it is not needed to provide information about the streams in the OD itself, since the OD can contain only an URL to an OD stored on a remote location, however keeping the same OD_ID. The OD is then retrieved from the remote location in a transparent manner, without the scene knowing its origin.

The ODs are conveyed to the terminal by using a simple command structure named OD Commands. By using these commands ODs can be inserted or removed from the scene in a dynamic manner. Moreover, each of the commands are timed, in the same way as the other streams (see Section 3.4), allowing them to tell the terminal when it should be prepared to receive another elementary stream. The usage of commands depends of the application and the user. For example, in a service that offers streaming of TV channels may use this feature to make new audio and video streams available. Because the information in ODs is vital to the application, it should be transported using a reliable channel. Moreover, in order to begin the MPEG-4 presentation, initial information about the scene and its object descriptors is transferred by using a special object descriptor named Initial OD (IOD). Besides the information in a regular OD, the IOD also contains information about the overall complexity of the scene, expressed in profile and level indications. Therefore the IOD should be transferred before any ES initialization, usually done in the session initialization stage.

I.3.2.3.3 Elementary Stream Descriptors

Elementary Stream Descriptors (ESDs) contain descriptive information about ES as illustrated in Figure 1.19.

Each elementary stream is identified by a unique ES_ID and an optional URL. The ES_ID is unique in a well-defined scope closely connected to the ODs. Therefore there

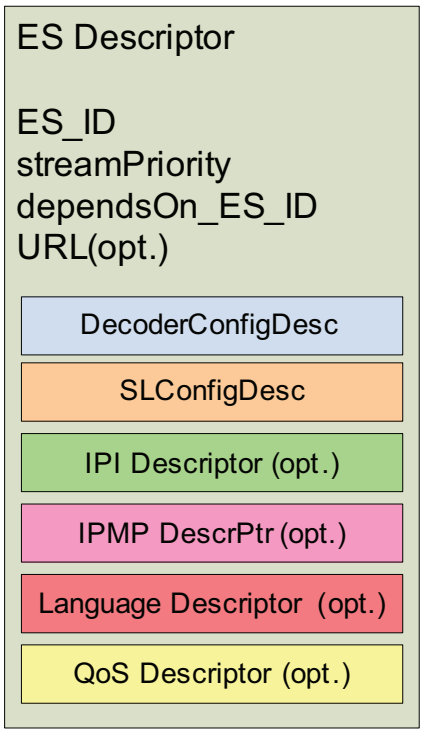


Figure 1.19: Elementary Stream Descriptor Contents

is a simple scoping rule: all scene description and OD streams that are associated to a single-object descriptor constitute a single name scope for the identifiers used by them, since they will all be attached to the scene through a single Inline node.

Contrary to the ODs, the URL field of the ESDs, if present, points to the stream itself, rather to the information about the stream. The mandatory DecoderConfig descriptor contains all the information needed to initialize the media decoder, including the object type, the average and maximum bit-rates, as well as the size required for the decoding buffer in the receiver terminal. Furthermore it may contain a binarized configuration data named Decoder Specific Info that is passed to the specified decoder. The type of the stream is identified by a minimum of 2 parameters: the first is the stream-type indication which specifies the general type of the stream (audio, visual, scene...) and the second is the object-type indication which specifies the exact compression scheme for the object. In the case of the AFX [8] standard, there is an additional identification number stored in the first byte of the decoder specific info that specifies the exact AFX stream type (e.g. BBA, 3DMC, FootPrint).

I.3.2.4 Synchronization Mechanisms

The synchronization of ES is done by using a system that includes time stamps and clock references, same as the one used in MPEG-2 systems. The discussion for this system in MPEG-4 can be separated in two main parts: the first is the System Decoder Model (SDM) [37] and the second is the synchronization layer, i.e., the sync layer.

I.3.2.4.1 System Decoder Model

The System Decoder Model defines the buffer and timing behavior of an MPEG-4 terminal. The SDM illustrated in Figure 1.20, unlike in MPEG-1 and MPEG-2, receives indi-

vidual ESs from the delivery layer trough a DMIF Application Interface (DAI). Choosing this kind of architecture allows the application to specify the network delivery protocol itself, independent of the standard. The only requirement that MPEG-4 exposes is the end-to-end delay, and it is up to the delivery layer to ensure it.

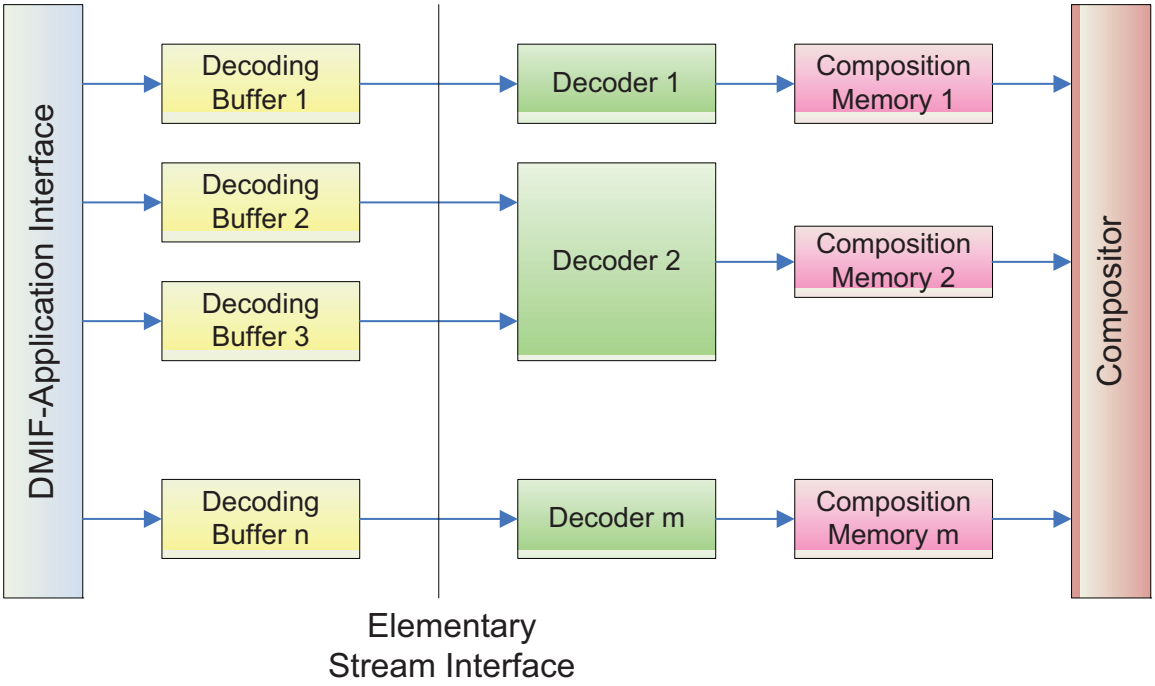


Figure 1.20: System Decoder Model

The core entity that is used trough the SDM is the Access Unit (AU). Each elementary stream is separated in a sequence of AUs, having their semantic meaning determined by the media encoders, thus unimportant for the SDM. However, there is one limitation: the AU is the smallest unit for which timing information can be associated because for each AU there are two timestamps, decoding and composition. The decoding timestamp specifies when the AU should be sent to the decoder, however it may or may not be present. The composition timestamp specifies when the decoded data should be available to the compositor. Often the decoding and composition timestamp have the same value, however the two values are retained because of some formats, for example video with bi-directional predictive coding where one frame has to be decoded before its display timestamp, so the frames that are in-between can be decoded.

By using this SDM, it is possible for the encoder to know how much decoding buffer resources are available at the client side at any time and for any stream, however it does not provide the possibility to know the size of the composition buffers.

The AUs that the DAI receive are first demultiplex and then sent to the appropriate decoding buffer. An AU is moved out of the decoding buffer when its decoding time arrives and then it is sent to the decoder. The decoder outputs the decoded data into the composition memory in the form of composition units. The relationship between the number of output composition units and the number of input AUs does not have to be one-to-one. For example, it is possible, as illustrated on Figure 1.20, to have one decoder that receives AUs from multiple decoding buffers and outputs into one single composition memory, therefore it is possible to generate one composition unit from multiple AUs. In addition, it is possible to generate more composition units from less AUs, for example

by interpolation in the case of object animation. Each composition unit is available for composition starting at the indicated composition time. The compositor accesses the composition memory at each rendered frame, and it is not concerned with the timing aspects of the composition unit.

I.3.2.4.2 The Sync Layer

The sync layer, located between the compression layer and the delivery layer, provides a flexible syntax that encodes all relevant information about an AU while it allows mapping of complete or partial AUs into the delivery layer protocol. The atomic entity, SL packet, of the sync layer is transferred to/from the delivery layer, however SL packets cannot be concatenated to obtain a parsable stream. This is done intentionally to avoid duplication of data by having the delivery layer that frames the packets. The purpose of SL packets is to allow the encoder to guide the fragmentation of the AUs. By knowing the size of the Maximum Transfer Unit (MTU), the encoder can provide self-contained packets that are smaller leading to better error resilience. For example, in the case of MPEG-4 video coding, SL packets correspond to video_packets, while AUs correspond to video_object_planes. Furthermore, smaller SL packets can be mixed into one delivery packet, like in the case of the FlexMux tool, therefore reducing the framing overhead.

The sync layer is designed with flexibility in mind, meaning that it can be configured separately for each elementary stream by using the SL descriptor in the ES descriptor. For example, different streams may require different timestamp precision, therefore different number of bits can be used for each, leading to lower bandwidth usage.

I.3.2.5 Scene-Graph Description

BIFS was introduced previously in section 2.5.1, therefore this section will introduce in more detail its features from the usage point of view.

I.3.2.5.1 Scene format

The scene in MPEG-4 BIFS [60] has two components: visual and audio. The visual part can be 2D, 3D or a combination of both, while the audio part can be a mix of many audio sources. As VRML, BIFS is also organized as a hierarchical structure or a scene tree, with the visible or audible object being the leaves of the tree or leaf nodes connected to branching points or grouping nodes. The tree structure allows to group nodes semantically, thus enabling organization of the scene in a manner convenient for easier management. For example, the translation of a leaf node is the result of the addition of all translation attached to any of its parents up to the root node.

BIFS currently implements more than 190 nodes [4]. However not all of these nodes are needed for all application, hence only a subset of these nodes has to be chosen to be used for these applications.

2D and 3D scenes One of the major features of BIFS in terms of composition is the ability to mix effectively 2D and 3D graphics primitives in one scene by using the *Layer2D* and *Layer3D* nodes. When the presentation engine encounters one of these nodes, it switches the rendering mode into 2D or 3D respectively. This allows having a 2D interface rendered on top of a 3D scene, e.g. a “heads up display” in the case of games.

The Shape node The *Shape* node is used to encapsulate a visible entity inside the scene. It has two main fields, *appearance* and *geometry*. The appearance field specifies the visual appearance of the entity, like its color or texture, and it points to an *Appearance* node. The Appearance node can point to a Material node type (*Material* or *Material2D* depending on the type of object) and/or to a Texture node. The geometry field specifies the shape of the entity, and it can point to different types of nodes, but, in our limited set, it points only to an *IndexedFaceSet* node.

Animation To support both animations, bone-based and morphing, BIFS integrates several nodes. The *SBVCAnimation* and *SBVCAnimationV2* are the main nodes, the latter being an improved version of the first. These animation nodes are in close relation to the BBA standard for animation compression. The URL field of the SBVCAnimation nodes point to the URL where the BBA animation is stored and the VirtualCharacters field hols one or more models animated by the same BBA stream.

Each of the VirtualCharacters is a *SBSkinnedModel* node holding information for the skeleton of the avatar (a tree structure of *SBBone* nodes) and its mesh is represented as a list of *Shape* nodes sharing the same coordinates and normals lists.

Another type of animation system includes Interpolators. Currently eight interpolator nodes exist: CoorindateInterpolator, CoorindateInterpolator2D, ColorInterpolator, NormalInterpolator, PositionInterpolator, PositionInterpolator2D, ScalarInterpolator and OrientationInterpolator. They are closely connected to the node or field type that they can update. The change of value is triggered by an event, e.g. time has changed, and therefore they are typically used to create small animations in the scene.

Referencing other scenes BIFS has support for including other scenes inside the tree, sub-scenes, enabled by using the *Inline* node. Therefore it becomes possible to have one MPEG-4 file having the structure of the scene, while the elements are located inside other MPEG-4 files. One advantage is that it is easy to add, remove and change content from the scene without significantly changing the size of the scene file itself, while keeping its complexity low. Another advantage is observable while streaming, where the elements of the scene can be easily saved and cached for further reference.

Script Scripting can be integrated into the scene by wrapping the script into *Script* nodes. The syntax of the script is an ECMA-Script (or JavaScript), allowing to have enriched interaction in the scene. In the case of games, it can be used to control the game logic on the terminal. Scripts can be activated by different means, including timed events and user actions. For example, a key press on the terminal can trigger execution of a script that changes the position of an object in the scene.

I.3.2.5.2 Scene Updates

One of most important BIFS assets is its dynamic capabilities. A BIFS scene is not static as a VRML scene, it can be updated in time. Since it follows the same transport means as other media types, the scene updates are packed in AUs, each containing one or more update commands.

Updating the data in a BIFS scene can be done in two ways: BIFS-Commands and BIFS-Anim. The next sections present both of them.

BIFS-Commands BIFS-Commands can update any part of the scene: node, node field, route and a single value in a multivalued field. There are three basic commands: Insert, Delete, and Replace, which are applied on the different parts of the scene.

The first command is always a *Scene Replace* command, that sets up the initial scene. The following commands can be a combination of any other commands. The execution time of the commands is the composition time-stamp field of the AU holding that command.

In a case of a static MPEG-4 file, all of the BIFS Commands are pre-created and executed in a sequence while playing the file. However in a streaming setup, they can be generated dynamically depending of different inputs, either from calculations or user input.

BIFS-Anim BIFS-Anim is the second mean of updating data in a scene. While BIFS-Commands allow updating any data, BIFS-Anim commands can update only numeric fields. BIFS-Anim commands are compressed and sent using either the intra or progressive mode. When using the intra mode, quantized absolute values are sent, whereas for the predictive mode, only differences between the current and the previous quantized values are sent.

I.3.2.6 Specific 3D Graphics Compression Tools

Being the most generic format, BIFS allows a generic compression of a scene-graph, however the downside is that it cannot completely exploit the redundancy of specific data such as meshes or animation curves. To overcome this, MPEG-4 defines methods for each kind of graphics primitive specified in Part 16 of the MPEG-4 standard [8] named Animation Framework Extension (AFX). The compressed data is stored in an AFX stream type, thus it has the same advantages as the other streams with regards to streaming and synchronization support. Since all methods use the same stream type, they are distinguished by using a type parameter stored in the first byte of the DecoderSpecificInfo descriptor in the ES Descriptor.

The methods can be separated in three main categories: for shapes, 3D Mesh Coding (3DMC) and Wavelet Subdivision Surface (WSS); for textures, Visual Texture Coding (VTC) as well as the native support of PNG and JPEG[2000]; and for animations, Coordinate (CI), Orientation (OI), and Position interpolators (PI), BBA, and Frame-Based Animated Mesh Compression (FAMC).

I.3.2.6.1 Geometry tools

The geometry tools compress 3D geometry stored in the scene. The geometry that is compressed is replaced by a *BitWrapper* node that can point to the compressed AFX stream, or contain the encoded data inside its buffer field.

Only three tools from this category are important for this research: 3DMC Extension, TFAN and MultiResolution Foot-Print Representations.

3DMC Extension [22] is used to compress a generic mesh stored into an *IndexedFaceSet* node by using efficiently coding the vertex data and its connectivity information. This compression encodes only one resolution of the mesh.

TFAN [49] is a 3D static mesh compression approach applicable directly to any 3D triangular mesh of arbitrary topology (ie, manifold or not, oriented or not, closed or open). It is based on partitioning of the mesh triangles into a set of triangle fans. An advantage

of TFAN is that the decoding is less complex than 3DMC, hence less processing power and resources are needed.

MultiResolution Foot-Print Representation (FootPrint in short) [21] is a coding tool for effective representation of city buildings. Since a city can have tens of thousands of buildings, rendering them at once will require significant processing power. Therefore the FootPrint tool proposes multiresolution representation of the buildings, based on their footprint (base of the building). This allows having a 2D representation of the city, as well as a 3D one created by extruding the footprint vertically by the specified height of the building and applying texture on the facade.

I.3.2.6.2 Texture tools

This group includes tools for specifying particular cases of textures including Depth Image-Based Representation and Multitexturing. The first one adds new nodes in the scene that can specify rendering of an image alongside depth information about each pixel. The second one adds new nodes that enable to integrate multiple textures for one object, thus providing the possibility to specify more complex features like bump mapping.

I.3.2.6.3 Animation tools

This section refers to tools that change or deform a mesh in a scene. There are two main categories: Deformation tools, and Generic skeleton, muscle and skin-based model definition and animation.

The deformation tools specify new nodes that encapsulate a mesh, thus modifying its form by a predefined function. The NonLinearDeformer tool specifies three types of deformations: tapering, twisting or bending. The FreeFormDeformation (FFD) tool wraps and deforms the mesh by using a NURBS surface.

Generic skeleton, muscle and skin-based model definition and animation tools specify new nodes used to represent a skinned skeleton model of a mesh that can be animated. The animation is encoded and compressed by a new stream named Bone Based Animation (BBA) [55]. The compression allows for effective streaming of the animation data.

I.4 Conclusion

In the first part of this chapter presented a short overview of the modern remote computing and how it changed from the simple to more complex applications. The different distributed applications for providing 3D content were analyzed and grouped by the technique they use to provide the content to the client in six major groups. The analysis was made based on the distribution of the work between the client and the server considering the requirements for game applications on mobile devices. It was observed that these techniques do not satisfy simultaneously all requirements. Therefore a new architecture is needed and it is propozed in the next chapters.

In the second part different standard multimedia scene description languages were analyzed. Then they were compared with respect to the supported features and compression and streaming capabilities. Since the MPEG-4 standard is the only one that has support for the needed features, it was chosen as the most appropriate for use in the architectures presented in this dissertation. Therefore it was analyzed in more detail, describing its targeted applications. Then it was described how streaming and synchronization are

handled by using ODs, ESDs and the Sync layer. Furthermore, the BIFS scene-graph representation features were presented, as well as some supported compression tools.

This chapter concluded the presentation of the state of the art in distributed architectures and rendering systems, as well as for scene-graph formalism. It was observed that at the current state, the mobile devices, as well as the mobile networks are capable of supporting distributed applications. The next chapters will present the main contribution of this thesis.

Chapter 2

Formal Framework for 3D Graphics Distributed Systems and Design of MPEG-4 Player Architectures

Abstract

This chapter presents the main contribution of this dissertation. The first part of this chapter proposes a formal framework that can be used to define and model distributed system architectures for rendering 2D and 3D graphics. First, a set of basic transformations are defined and all architectures that were presented in the state of the art (Section I.2) are modeled using the proposed framework. Then a new distributed architecture is presented based on these transformations. All components of the architecture are analyzed, and a solution for each of them is proposed.

The most significant part of the new distributed architecture is the MPEG-4 based player on the client side, and therefore the second part of this chapter explores and proposes an architecture design for MPEG-4 player optimized for a powerful platform. The player architecture extends the System Decoder Model (SDM) proposed by the MPEG-4 standard. A set of requirements are presented and solutions are proposed for each of them. It is observed that designing an MPEG-4 player can be a difficult task, hence a framework for accessing MPEG-4 content is proposed in the third part of this chapter. The framework uses a simple interface for obtaining the 3D related data.

In order to complete the distributed architecture, the last part of this chapter proposes a simplified version for MPEG-4 player on a mobile device. Considering that mobile devices are less powerful than personal computers, the player architecture has to be adapted. Furthermore, additional optimizations, made in order to decode and display the content with the best performances, are presented.

I.1 Introduction

As analyzed in Chapter 1, none of the existent distributed systems satisfies our requirements simultaneously. Furthermore, it can be observed that most of them use different representation formalisms to describe the architecture, hence they can not be analyzed theoretically on a common ground.

Therefore, in order to analyze them, we first define a theoretical representation. To accomplish this, the systems are divided in several connected processing blocks that transform the input data from one form into another. These blocks have to be chosen carefully in order to be able to utilize them for modeling each system that was presented. The model will help to better understand the operation and use-case scenario, as well as their limitations.

I.2 Formal Framework Definition

The process of a distributed system can be represented as a work-flow involving a set of transformations. The basis of this work was done previously by Preda et al. in [56] where they analyzed systems that visualize 3D graphical models, possibly coming from a server across a network channel. The limitation in their work is that the modeling process takes into account only one 3D object, however complex distributed 3D applications, e.g. games, integrates many 3D objects and other media of different complexities.

In the mathematical model proposed in [56], each object is represented as a set of features $\{F_i\}$ that is the input of each basic function. To accommodate more complex applications, in the model presented in this chapter, the input of the process is the scene-graph SG^t, which is defined as a set of nodes $\{N_i^t\}$ at a time t. The nodes can be grouping nodes GP that are references to other nodes, data nodes GI that hold renderable data, or scene control nodes SC that influence the scene. The scene-graph is defined in Equation 2.1.

$$SG = \left\{N_i^t\right\} = \left\{GP_i^t, GI_i^t, SC_i^t\right\} \tag{2.1}$$

The output of the model remains the same, $\{P_i\}^{2D} = \{R_i, G_i, B_i, A_i\}$, a set of 2D pixels, with color RGB and transparency (A), ready to be displayed; or $\{P_i\}^{3D} = \{R_i, G_i, B_i, A_i, D_i\}$, a set of 3D pixels containing a depth component computed at rendering time. The goal is to produce either one of the two sets $\{P_i\}^{2D}$ or $\{P_i\}^{3D}$ from the scene-graph SG^t in an optimized way, taking into account the constraints coming from the components in the system.

In [56] the model is based on four transformations: rendering, coding, simplification and modeling. However, for more complex application, and updated model is needed and therefore we extend this model to include one more transformation: scene-graph updates. This transformation models the part of the architecture that modifies the scene according to the users commands or other types of input. This is an important transformation because depending on how the scene is updated, it can lead to different optimizations. The modifications is done by updating the state of the different objects in the scene, as well as adding new objects or removing existing ones. Furthermore, the definition of the four transformations has to be updated to consider the scene-graph in their processing.

The next section presents all transformations and describes their operation in details.

I.2.1 Set of Transformations

The possible transformations in the process are: rendering, coding, simplification, modeling and scene updates. Their mathematical definition is defined as follows:

$$Rendering : \{P_i\} = R(SG) \tag{2.2}$$
$$Coding : \left\{SG_i^C\right\} = C(SG) \tag{2.3}$$
$$Simplification : \left\{SG_i^S\right\} = S(SG) \tag{2.4}$$
$$Modeling : \{SG_i\} = M(SG) \tag{2.5}$$
$$Scene - updates : SG^t = U\left(SG^{t-1}, \{I_i\}\right) \tag{2.6}$$

where $\{I_i\}$ represents the different inputs in the game.

I.2.1.1 Rendering

Rendering R, which projects the model onto planes suitable for display, is converting SG into $\{P_i\}$ as defined by Equation 2.2. R wraps the graphics card API and it is used to initialize the graphics rendering on the client terminal or the server. Therefore the input SG is a collection of graphics objects represented in a format that can be directly consumed by the graphics card. R is one of the components that requires a lot of processing power and therefore it has to be optimized.

I.2.1.2 Coding

Coding C is used to make the representation of the scene-graph compact, as defined in Equation 2.3. It is a combination of three transformations:

1. Encoding (compression)
2. Transmission (in a distributed environment)
3. Decoding (decompression)

The encoding process compresses the data into a more compact format. The compression method depends on the type of the input data. For example, the encoding process transforms the scene-graph into a tight binary representation, encoding each node and data separately, thus different encoding methods are used for different node types. The mesh geometry data (3D positions of vertices) is less sensitive to errors, therefore a lossy compression scheme can be used, while the scene-graph information should be exactly the same, therefore a lossless compression scheme is used. This allows having better compression results than when using a common compression method for all data. It also allows making the representation more robust against channel errors.

The decoding process transforms the data back into the original format, thus in the case of a scene-graph, it restores the original structure exactly as it was. On the other hand, if the data is compressed using a lossy compression, this is not possible, thus similar data is restored. It is mostly noticeable in the case of video encoding, where blocking artifacts can be observed from low bit-rate encoding.

I.2.1.3 Simplification

In many cases it is possible to simplify the set of features without affecting the final quality of the presented pix-map, because the display resolution, the viewing conditions or the scene allow the elimination of irrelevant information. This operation could be performed simultaneously with rendering, but it is advantageous to model it as a separate transformation, called simplification S, which provides adaptation to terminal and viewing constraints, as indicated by Equation 2.4.

The display resolution has the biggest influence in the case where video should be transferred, thus a video with greater resolution than the one on the terminal will look very similar to a video with the same resolution as the terminal. The viewing condition can have impact on the order in which the scene objects are loaded (or transferred), thus objects that are inside the viewing frustum of the came can have greater priority than the ones that are outside. This will significantly improve the user experience, since the delay of the loading of the scene will be virtually reduced. It is also possible that the organization of the scene itself leads to simplification, as for example, if an object is always far from the camera, its details can be reduced, including the number of vertices and the details of its texture, without sacrificing the visual impression of the scene.

I.2.1.4 Modeling

The fourth transformation, Modeling M (Equation 2.5), transforms the scene-graph representation from one formalizm into another. This operation can be simple or complex depending on the input and output data. Apart from being a direct transformation, M can be a combination of two other operations, Rendering and Vectorization. Vectorization is defined as reconstruction of a scene-graph from pixels (Equation 2.7).

$$Vectorization : SG = V(\{P_i\}) \tag{2.7}$$

$$\tag{2.8}$$

Vectorization usually involves a process of applying a feature detection algorithm on a rendered image in order to reconstruct the scene-graph. However the scene-graph does not have to be in the original representation formalism, hence the modeling asspect of the vectorization.

I.2.1.5 Scene Updates

A part from the case where there is only one object, in a game the state changes continuously, being affected by different inputs. The scene update transformation U represents these updates, depending of the previous state of the scene-graph SG^{t-1} and a set of influences $\{I_i\}$, as it is defined in Equation 2.6. However, it does not mean that the whole scene is updated: only one node may be affected by the transform. The calculation of U depends of the game itself and its complexity does not reflect the number of nodes affected, e.g. a complex calculation like the Artificial Intelligence (AI) for a chess game can produce change in only one node, e.g. move a pawn.

In the case of game, the inputs can be of very different nature:

- User actions - key press, mouse movement that result in update in the scene-graph

- Multiplayer action from a remote user - update a position of a remote player in the local game state
- Artificial Intelligence - movement of AI entities inside the game
- Script - predefined movement of entities
- Scene description - structure of the initial scene-graph in the game
- Physics-based simulation - movement and interaction of entities in same way as if they were real objects

By using the previously defined formal framework, the next section analyzes the techniques presented in the state of the art on a common ground.

I.3 Analysis using the Formal Framework

Section I.2 proposed a formal framework for describing distributed system architectures for rendering 2D and 3D graphics. This framework allows to represent and analyze the architectures analyzed in the state of the art on a common ground. The order of transformations is very flexible, and capable to model different architectures. Furthermore, not all transformations are needed for all architectures.

By using the transformations, some architectures are described and defined in the next part of this section. The following equations represent the mathematical description of the architectures:

$$\{P_i\} = R \circ C \circ M \circ U\left(SG^{t-1}, \{I_i\}\right) \tag{2.9}$$
$$\{P_i\} = C \circ S \circ R \circ M \circ U\left(SG^{t-1}, \{I_i\}\right) \tag{2.10}$$
$$\{P_i\} = R' \circ M' \circ C \circ V \circ R \circ M \circ U\left(Sg^{t-1}, \{I_i\}\right) \tag{2.11}$$
$$\{P_i\} = R' \circ M' \circ C\left(\left\{U\left(\{GI, SC\}^{t-1}, \{I_i\}\right), V \circ R \circ M \circ U\left(\{GP\}^{t-1}, \{I_i\}\right)\right\}\right) \tag{2.12}$$
$$\{P_i\} = R \circ M \circ U\left(SG^{t-1}, \{I_i\}, C \circ S\left(\{G_d\}\right)\right) \tag{2.13}$$
$$\{P_i\} = R \circ M \circ C \circ S \circ U\left(Sg^{t-1}, \{I_i\}\right) \tag{2.14}$$

Equation 2.9 models the graphics commands solution presented in Section 1.2.1, where at the servers side the commands sent from the program to the graphics card are intercepted, and sent to the client who forward them to its own graphics card. The transformations are performed in the following order:

1. The scene-graph is updated by U.
2. The scene-graph is modeled into a new scene-graph formalism (SG') that uses a graphics card representation by M.
3. SG' is compressed and sent to the client, where it is decompressed by C.
4. SG' is rasterized to a pixel buffer by R at the client side and it is displayed.

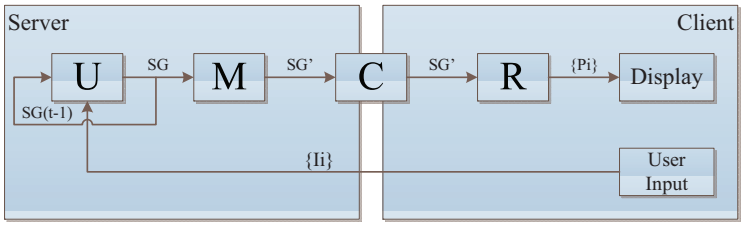


Figure 2.1: Model of graphics commands based solutions

This is illustrated in Figure 2.1 where Fb is a flat buffer of the graphics primitives, including also textures, shaders and other information needed for rendering.

Equation 2.10 models the pixel based solution presented in section 1.2.2.1 and 1.2.2.2, where the scene is rendered on the server and pixel buffers are sent to the client for direct rendering. The difference between the two methods is the type of data that is transferred. The transformations are performed in the following order:

1. The scene-graph is updated by U.
2. The scene-graph is modeled into a new scene-graph formalism (SG') that uses a graphics card representation by M.
3. SG' is rendered into a pixel buffer by R.
4. The pixel buffer is simplified (scaled to fit the client screen) by S.
5. The simplified pixel buffer is compressed depending of the technique, transferred to the client where it is decompressed by C and displayed.

This is illustrated in Figure 2.2.

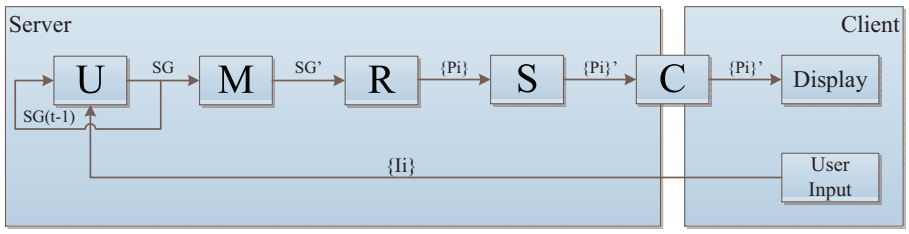


Figure 2.2: Model of pixel based solutions

Equation 2.11 models the 2D primitives based solution presented in section 1.2.3.1, where the scene is rendered on the server, the feature lines are extracted, and sent to the client for rendering. The transformations are performed in the following order:

1. The scene-graph is updated by U.
2. The scene-graph is modeled into a new scene-graph formalism (SG') that uses a graphics card representation by M.
3. SG' is rendered into a pixel buffer by R.
4. The pixel buffer is analyzed by an edge detection algorithm and 2D lines are extracted by V.

5. The 2D primitives are coded and transferred to the client by C.
6. The scene-graph is modeled into a new scene-graph formalism (SG''') that uses a graphics card representation by M'.
7. SG''' is rendered on the client side by R'.

This is illustrated in Figure 2.3.

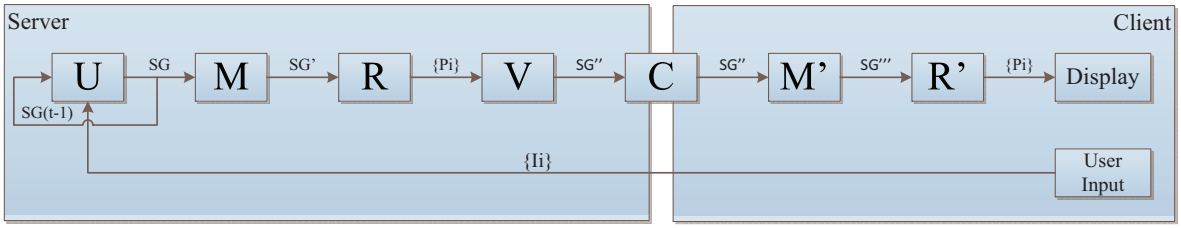


Figure 2.3: Model of 2D primitives based solutions

Equation 2.12 models the 3D vectors based solution presented in 1.2.3.2.1, where the 3D objects are transformed in 3D vectors and sent to the client. The transformations are performed in the following order:

1. The scene-graph is updated by U.
2. The data nodes of the scene-graph are modeled into a new data nodes formalism (GP') that uses a graphics card representation by M.
3. GP' are rendered into a pixel buffer by R.
4. The pixel buffer for each data node is analyzed by an edge detection algorithm and 3D lines are extracted by V.
5. The new scene-graph (SG') is compressed and transferred to the client by C.
6. SG' is modeled into a new scene-graph formalism (SG'') that uses a graphics card representation by M'.
7. SG' is rendered at the client by R'.

This is illustrated in Figure 2.4, where Li is a set of 3D lines representing each 3D object separately.

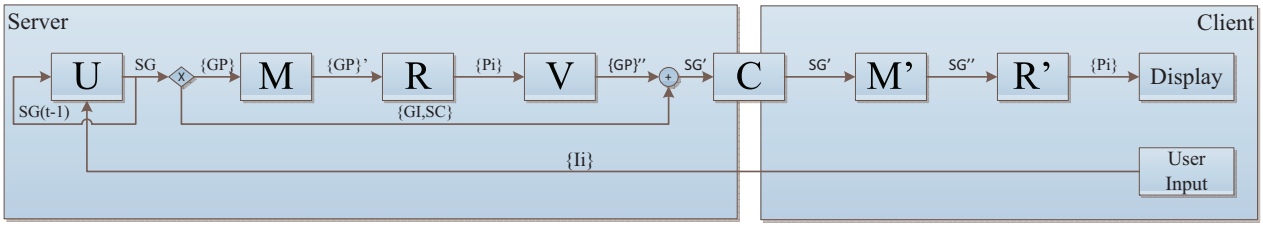


Figure 2.4: Model of 3D primitives based solutions

Equation 2.13 models the single object based solutions presented in section 1.2.3.2.2, where optimization and compression is executed per object. The transformations are performed in the following order:

1. The object is simplified by S.
2. The simplified object (GP') is compressed and transferred to the client by C.
3. GP' is integrated in the scene-graph and updated by U.
4. Ihe scene-graph is modeled into a new scene-graph formalism (SG') that uses a graphics card representation by M.
5. SG' is rendered at the client by R.

This is illustrated in Figure 2.5.

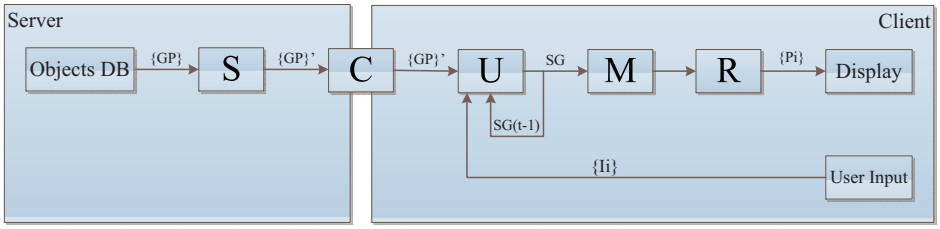


Figure 2.5: Model of single object adaptation based solutions

Equation 2.14 models the multiple objects based solution presented in section 1.2.3.2.2, where an optimization is done by reducing the scene-graph. The transformations are performed in the following order:

1. The scene-graph is updated by U.
2. The structure of the scene-graph is simplified (i.e. nodes are removed temporary for sending) by S depending of different conditions and parameters.
3. The scene-graph is compressed and transferred to the client by C.
4. Ihe scene-graph is modeled into a new scene-graph formalism (SG') that uses a graphics card representation by M.
5. SG' is rendered at the client by R.

This is illustrated in Figure 2.6.

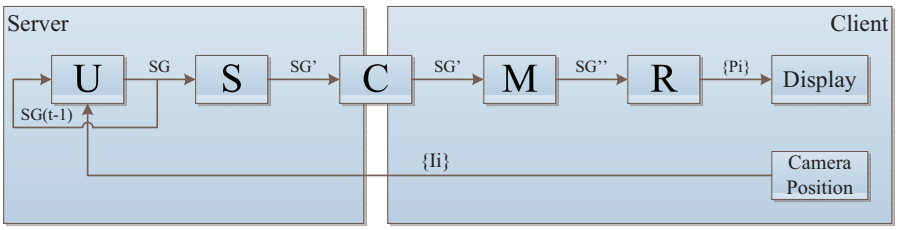


Figure 2.6: Model of multiple object adaptation based solutions

From the previous examples it can be observed that the defined transformations are sufficient to describe distributed architectures. The list of examples represents all architectures that were analyzed in the state of the art (Section I.2). However this formalism

is not limited only to those architectures, on the contrary, the intention is to use it to represent any distributed graphics architecture that could be developed in the future.

All these models are in principle capable of producing the same output, or at least outputs that are visually indistinguishable from each other. Which one has to be used depends on the constraints of the environment in the specific case: computational power available on the server, available bandwidth, network latency and computational power on the client side.

Terminals with very low processing power will be best served by the architecture described in (2.10), where the client requirements are only to be capable to decode and display the pixmap. If the computational power of the client increases, it becomes possible that it can do rendering, therefore it is possible that instead of pixels, objects can be transferred (2.11), thus less bandwidth may be used. As a special case is a terminal which has graphical capabilities but not enough processing power for other tasks, thus the processing is done on the server and graphical commands are transferred to the client (2.9). As the computational power increases again, it will become possible that it can render simple 3D objects like 3D vectors (2.12) and further increase will enable rendering of more complex 3D objects, i.e. meshes (2.13). This means that the load on the server will decrease, however the data has to be simplified to fit the clients capabilities. Sometimes the size of the scene-graph is too big that even data simplification is not enough to enable smooth experience, thus a simplification of the scene-graph is needed, as defined in (2.14) It means that not all nodes are transferred to the client, thus only the ones that are necessary, for example only the objects that are inside the camera frustum.

For designing a real system, each of these architectures should be tested and evaluated using those system constraints, thus a decision which one to use will depends of the analysis.

This thesis proposes a system where playing complex 3D games on mobile phones. Since the mobile phones have low processing power, the system proposed in (2.10) seems the most appropriate, however the bandwidth requirements for having a visually satisfying experience are too big. Because the mobile devices are powerful enough to handle rendering of 2D and 3D graphics, therefore some of the other systems may be used. The system described by (2.9) needs the same graphics hardware as a PC, however since the hardware is not available on a mobile phone, it cannot be used. The system defined in (2.11) requires less bandwidth by sending only 2D vectors to the client, however they need to be sent for each frame and their visual appearance is not satisfying for a game. The architecture defined in (2.12) is more bandwidth effective, thus sending the 3D vectors for each object only once, however the visual appearance remains similar. The architectures defined in (2.13) are intended for systems where a single object is transferred, hence they do not handle systems with a complex scene-graph, although they may be used as a part of such system. Therefore the systems defined with (2.14) seem the most appropriate, being capable of rendering visually satisfying images and being capable of handling complex scene-graphs, however, the limitation is that the user can only control the position of the camera, hence it is not usable as a game architecture, where more sophisticated user actions are needed.

The previous analysis confirmed that none of the systems presented in the state of the art are capable of supporting game applications with the given requirements. Therefore, now that the formal framework is defined, it can be used to design a new system that satisfies all requirements simultaneously.

I.4 Design of a new Distributed System Architecture

The system that is modeled is a distributed one capable of supporting game applications for mobile devices. Using the formal framework, the system equation can be defined as follows:

$$\{P_i\} = R \circ C \circ S \circ U \left(Sg^{t-1}, \{I_i\}\right) \tag{2.15}$$

The transformations are performed in the following order:

1. The scene-graph is updated by U.
2. The structure of the scene-graph is simplified (i.e. nodes are removed temporary for sending) by S depending of different conditions and parameters.
3. The scene-graph is compressed and transferred to the client by C.
4. Ihe scene-graph is modeled into a new scene-graph formalism (SG') that uses a graphics card representation by M.
5. SG' is rendered at the client by R.

Since in every game the content is organized in some sort of scene-graph, it is logical to conclude that using the same one for all games will increase the effectiveness of their development. As it was demonstrated in Chapter 1, there are already some scene-graph formats available, hence developing a new one is not efficient. The MPEG-4 standard was proposed as the most features complete for a game architecture, therefore it will be used as the scene-graph format for the proposed distributed architecture.

The second standardization can be done on the type of the data sent from the client to the server. For example in a car racing game holding the key for forward pressed increases the forward acceleration, however in another game the same forward key may only change the position of the viewer for a constant value, hence a way to standardize this data is needed. The easiest and most obvious solution is not to send the action that the key produces, but the code of the key itself, hence the logic on the server will interpret it and produce the desired effect. This relieves the client application of any game specific calculations, thus enabling to use the same one for different games. Since not all terminals have the same keys, the server needs to be informed of their availability on the client. This can be done in the initialization stage of the game where the client will send information for the type of the terminal to the server.

Standardizing the user input and the scene-graph format enables having client independence from a specific game. It was concluded in Chapter 1 that the MPEG-4 standard has the needed features, therefore the player will be a standard MPEG-4 player. However, not all features of the standard need to be supported for implementing a game architecture.

The block diagram of the proposed distributed architecture is displayed on Figure 2.7. In the next section we propose a solution for each component of the architecture.

For rendering the scene-graph, the nodes that need to be supported are presented in Table 2.1. Detailed descriptions of each of these nodes can be found in the MPEG-4 standard. The reduction of the number of nodes helps in optimizing the rendering engine, as well as in reducing the size of the code, i.e. the size of the application's executable file.

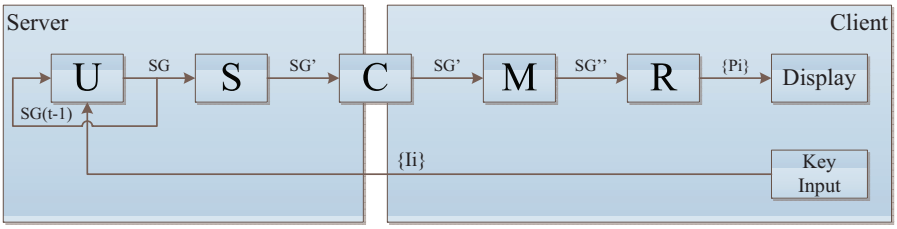


Figure 2.7: Proposed Architecture

Table 2.1: Subset of BIFS nodes

Node Name	Node Name
Appearance	TextureCoordinate
Background2D	Transform
Coordinate	Transform2D
FontStyle	Valuator
Group	Viewpoint
ImageTexture	InputSensor
IndexedFaceSet	BitWrapper
Inline	SBBone
Layer2D	SBSegment
Layer3D	SBSite
Material	SBSkinnedModel
Material2D	SBVCAnimation
NavigationInfo	MorphShape
Normal	SBVCAnimationV2
OrderedGroup	Rectangle
Shape	Switch
Script	DirectionalLight
Text	PointLight

Furthermore, it allows faster development of applications because the learning curve for the MPEG-4 standard is shortened. A number of nodes allows for optimization at the server side, thus a speed-up for the generation of the scene-graph and its encoding can be achieved.

As it was described in Chapter 1, updating the scene can be done using two techniques: BIFS commands and BIFS-Anim. The BIFS update commands can change any part of the scene, while the BIFS-Anim commands can update only limited number of fields (e.g. SFVec3f, SFRotation, etc.). In the case of games, the scene may change not only in the position of the different assets, but also new assets may be added or removed, hence the BIFS commands mechanism has to be integrated. On the other hand, the BIFS-Anim mechanism offers better compression for updating position of assets. Since the BIFS update commands include the same features as BIFS-Anim, and furthermore to reduce the complexity of encoders and decoders, they will be the only ones that are supported by the distributed architecture. The MPEG-4 standard does not define the transport protocol, hence it can be chosen at will. For this architecture, the RTSP standard was selected as most appropriate because of its integrated support for the MPEG-4 standard.

Sending key code to the server can be achieved by two means: using the ServerCom-

mand node or using an AJAX request from a Script node. Using a ServerCommand node implies defining an upStream channel using an ESDescriptor that enables the upstream tag. This descriptor must depend on a downstream ES, in this case the BIFS stream. Furthermore, it means that the server and the client have to implement support for an upstream channel, thus adding further complexity to the code and to the learning curve. The AJAX request is already a built-in feature in the JavaScript code of the Script node, hence there is no need to add this feature to the client. The key code of the pressed key is integrated in the request header, therefore the server need only to integrate a simple parsing component to read it. In terms of communication complexity, the upstream communication is lighter since it does not have to sent HTTP headers as the AJAX request. However the bandwidth is not an issue because the user commands come infrequently, not more than two in one second, and the current network capacity is far greater than that. Furthermore, the user commands are important data, hence they have to be transferred in a secure manner. One solution is to use a TCP connection for the upstream channel, like the one used by AJAX request. Therefore, we can conclude that in this configuration the AJAX request is a better choice than the ServerCommand.

Each component in the distributed architecture can be associated with the corresponding technology. The scene-graph is based on the MPEG-4 scene-graph format, i.e. BIFS. The scene updates component U depends of the game itself, however the input is received from the client using an AJAX (i.e. HTTP) request. The Simplification component S may simplify the scene-graph by removing nodes that are not visible from the current camera position and produces a simplified scene-graph Sg'. The coding component processes the scene-graph updates and generates a BIFS update command that is transferred trough the network using RTSP to the client where it is decoded and applied to the local BIFS scene. The Renderer component R parses the BIFS scene, and displays it.

Considering the previously discussed design of the new distributed architecture, the next section presents its implementation.

I.4.1 Implementation of the new Distributed System Architecture

Using BIFS for streaming 3D content has been explored by Hosseini M.[38] proposing a powerful JAVA-based framework, however less appropriate for mobile devices. Using BIFS for games is not completely new, being already exploited by Tran S. M. [62] who proposed several 2D games. However, the updates are handled locally, no server being involved. To our knowledge, no other system addressing 3D games with client-server interaction using BIFS has been published.

In addition to representing graphics assets and scenes locally, MPEG-4 introduced a mechanism to update the scene from a server. BIFS-Commands are used to modify a set of properties of the scene at a given time, being possible to insert, delete and replace nodes and fields, as well as to replace the entire scene. The following section introduces a client-server architecture and a communication protocol dedicated to mobile games exploiting the remote commands feature of MPEG-4 BIFS.

I.4.1.1 Requirements

Two main categories of requirements have driven our developments in proposing the client-server architecture:

- for game creators, the deployment of a game on a large category of terminals should not lead to additional development cost,
- for players, the game experience (mainly measured in game reactivity and loading time) should be similar compared with a game locally installed and executed.

The main idea is to separate the different components presented in a traditional game into components that are executed on the server and components that are executed on the terminal. In a simplified scheme as the one illustrated in Figure 2.8, one may observe two major high processing components in a game: the game logic engine and the rendering engine.

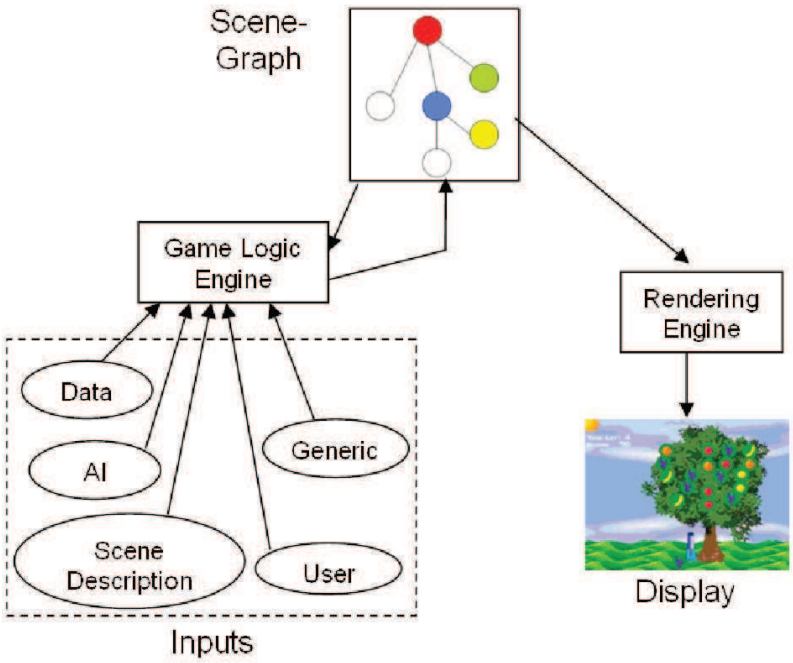


Figure 2.8: The main functional components of an arbitrary game

The first one receives updates from modules such as data parsers, user interaction manager, artificial intelligence, and network, and updates the status of an internal scene graph. The second is in charge of synthesizing the images by interpreting the internal scene graph.

I.4.1.2 Description of the Proposed Distributed System Architecture

As shown in Chapter 1, MPEG-4 has the capability of representing (in a compressed form) a scene graph and graphics primitives and an MPEG-4 player is able to interpret them to produce the corresponding synthetic images. The main idea proposed here is to replace the rendering engine of the game (right side in Figure 2.8) with an MPEG-4 player, with the following consequences: during the game, the scene graph (or parts of it) has to be transmitted to the client and the user input (captured by the client) has to be transmitted to the server. Figure 2.9 illustrates the proposed architecture. The direct advantage is that the MPEG-4 player is a standard player and does not contain any game specific code.

The main underlying idea of the distributed architecture proposed in Figure 2.9 is to execute the game logic on the server and the rendering on the terminal. Therefore,

different types of games may be rooted in the proposed architecture. The only strong requirement is the latency allowed by the game play. According to the games classification with respect to complexity and amount of motion on the screen, as proposed by Claypool M. [25], and game latency [26], our goal is to test the distributed architecture and state on its appropriateness and the usage conditions for the following four classes of games: third person isometric, omnipresent, third person linear, and first person.

In addition, the player receives only what is necessary at each step of the game (interface 1 in Figure 2.9). For example, in the initial phase only some 2D graphics primitives representing the menu of the game are transmitted. The 3D assets are sent only when they are used, the MPEG-4 compression ensuring fast transmission.

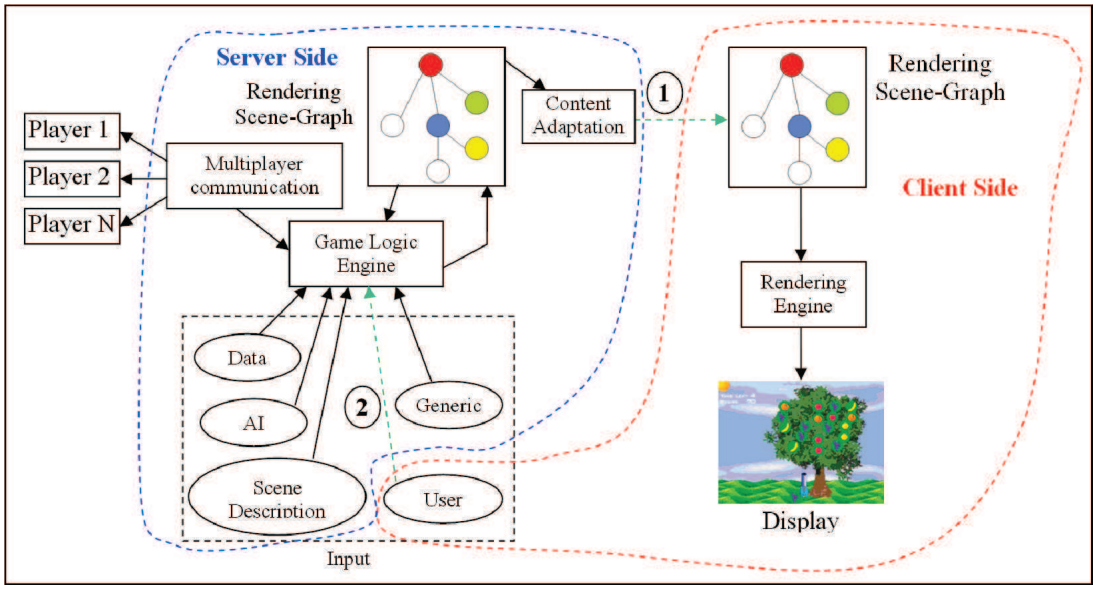


Figure 2.9: Proposed architecture for mobile games using an MPEG-4 player on the client side

During the game-play, the majority of the communication data consists in updates (position, orientation) of assets in the local scene. Let us note that for games containing complex assets it is also possible to download the scene graph, i.e. an MPEG-4 file, before starting playing. The off-line transfer of content has a similar functionality to the caching mechanism proposed by Eisert P. [31]. In addition, it is possible to adapt the graphics assets for a specific terminal [50], allowing for the best possible trade-off between performance and quality.

In the proposed distributed architecture, the communication characterized by interfaces 1 and 2 in Figure 2.9, unlike in [31], is based on a higher level of control: the graphic primitives can be grouped and controlled as a whole by using few parameters. The MPEG-4 standard allows any part of the scene to be loaded, changed, reorganized or deleted. For example, the same game can be played on a rendering engine that supports 3D primitives, most probably accelerated by dedicated hardware, and simultaneously on a rendering engine that only supports 2D primitives.

I.4.1.3 Analysis of the Proposed Distributed System Architecture

This flexible approach allows the distribution of the games on multiple platforms without the need to change the code of the game logic. Another advantage is the possibility to

improve the game logic without additional costs to the client, allowing easy bug-fixing, adding features and different optimizations.

Since the game logic can be hosted on a server that has much better performances than the client terminal, it can implement more advanced features, traditionally not supported on the client terminal. These features may include advanced artificial intelligence or advanced physics. These improvements will not change anything on the user side, therefore allowing the user to play more complex games on the same terminal.

The flexibility of the proposed distributed architecture makes it also appropriate for powerful terminals with the goal of reusing the rendering engine between different games. Under these circumstances, it is possible to download the game logic server software, install it on the terminal and use it to play the game locally. However, in order to maintain this approach effective, the games should be designed from the start bearing in mind the requirements and the restrictions.

The rendering engine (ensured by the MPEG-4 player) can be integrated into the firmware of the device (i.e. supplied with the device), allowing the manufacturer to optimize the decoders and rendering for specific hardware. Let us note that, besides BIFS, which is a generic compression algorithm able to reduce the data size up to 15:1, additional tools exist in MPEG-4 improving the compression of graphics primitives up to 40:1 [42].

Table 2.2 summarizes the advantages and disadvantages of the proposed approach with respect to the local game play.

It is straightforward to observe from Table 2.2 that the main drawback of the proposed method is the sensitivity to the network latency. Another weak point is the adaptation of existing games. Two components need to be considered: sending the 3D content to the client and getting the user commands from the client. There are three possibilities for conveying the content to the client:

- If the source code of the game is accessible, then it can be relatively easily modified for fitting with the proposed distributed architecture
- If a high level API is available for the game, supporting access to the graphics objects and the scene-graph, then a separate application can be created that will connect the architecture and the game
- If no game API is available, then a wrapper for the 3D library that is used by the game can be created. It will extract the geometry and transformations from the graphics calls. The cost is the need to implement a converter between graphics commands and BIFS updates

There are two possibilities for receiving the user commands:

- If the source code of the game is available, it can be modified easily for receiving the commands
- Otherwise, a software module can be created that will receive the commands from the client, create the appropriate system messages and send them to the game

The most important part of the new distributed architecture is the game client, i.e. the MPEG-4 player, that is executed on the users terminal. It has to be designed carefully considering all requirements. Therefore the next section presents the design of the MPEG-4 player for a powerful platform.

Table 2.2: Comparative evaluation of the proposed method with respect to a game executed locally

	Local game play	Proposed architecture
Advantages		
Software development	The game must be compiled for each type of terminal	The game logic is compiled once for the dedicated server. Game rendering is ensured by a standard MPEG-4 player, optimized for the specific terminal
Advanced game features	Reduced due to the limitation of the terminal	By choosing high end servers, any modern game feature, such as advanced Artificial Intelligence can be supported
Rendering frame rate	Since the terminal processing power is shared between game logic and rendering, it is expected that the frame rate is smaller	High because the terminal only performs rendering and asynchronous scene graph decoding
Game state consistency in multi-player games	Synchronization signals should be sent between terminals and complex schemas must be implemented	Synchronization is directly ensured by the server that controls the scene graph of each terminal at each step
Security	Games can be easily copied	The game starts only after connection to the server, where an authorization protocol may be easily set up
Maintenance and game updates management	Patches should be compiled for each version of the game	Easily and globally performed on the server
Disadvantages		
Network	No impact	The game cannot run without network connection
Network latency	No impact	The game experience decreases if the loop (user interaction server processing update commands rendering) is not fast enough[1]
Adaptation of existent local games to the proposed distributed architecture	No impact	Access to the game source code is recommended, but it is not necessary

I.5 Design of an MPEG-4 Player Architecture for Powerful Platforms

This section presents a player of MPEG-4 3D content that is designed in a way that it optimizes the MPEG-4 System Decoder Model (SDM) to use the processing power of multi-core CPUs to deliver better, fast and smooth user experience, and extend its features to include new scene-graph formats.

The design of the optimized SDM is performed in several steps: the first step specifies the requirements that need to be satisfied, the second step is the design of the new player architecture that satisfies the requirements and the last one is the implementation.

I.5.1 Requirements

Since the player is based on the MPEG-4 standard, it is logical to derive the requirements from the standard itself, however not all of its features need to be supported.

One of the strongest attributes of the MPEG-4 standard is its support for streaming, hence all data is organized accordingly. However, the standard only defines the structure of the packets, and not the underlying transport protocol. Therefore, to support multiple transport protocols (loading from a local MPEG-4 file can be viewed as one), it is necessary to separate the streaming protocol from the other parts of the player architecture (i.e. decoders and compositors), and having only SL packets (see Section I.3.2.4.2) as communication data. Furthermore, all data needs to be synchronized and composed at the right time, as defined by their time-stamps.

From its creation, the MPEG-4 standard uses BIFS as scene-graph and composition format. However, with the introduction of Part 25, other formalisms based on XML (eXtensible Markup Language) can be used as scene-graph formats. However, not all of them are compatible with the BIFS organization structure, therefore it is necessary to separate BIFS from the composition engine.

The MPEG-4 standard integrates several compressed formats for different types of data. An optimal implementation should enable easy integration of decoders, taking into account that some of them are used for the same type of data (e.g. PNG and JPEG2000 for image).

One important necessary feature is to be able to integrate the player in a website, enabling Internet experience of playing MPEG-4 content. However, usually there are different web browsers that are used for viewing web pages, and in most of them there is a different way of integrating a plug-in in the page, hence a different version is needed for each of them. Furthermore, it should be possible to be able to run the player as a standalone application, as well as to run it on different operating systems. Therefore, the visual interface[2] should be easily separated from the rest of the application.

From the previous discussion, the following main requirements are derived:

1. Receive data from multiple steams, either from local storage devices or network;
2. Utilize optimally the capacity of the client terminal;
3. Synchronize all data streams;
4. Accommodate multiple scene-graph formalisms;

[2]The visual interface is a part of the application that receives user input from mouse and keyboard, and displays the window of the application.

5. Easily integrate different decoders;

6. Easily integrate in different applications and web browsers.

The first requirement is satisfied by the original SDM by implementing the DMIF interface.

The second requirements is not specified in the SDM, hence it is up to the player architecture implementation to address it.

The third requirement is not completely satisfied by the original SDM. While the synchronization between the video and audio streams is not very complex, rendering a 3D scene synchronously has some difficulties because compositing the scene requires many resources including 3D meshe, textures and animation data. All of these resources have to loaded on the graphics card, and doing this optimally requires using only one thread for the loading process. Furthermore, the mesh and its animation data are closely connected in a sense that the mesh cannot be rendered without the animation data.

The fourth requirement is not satisfied by the original SDM because it uses only one scene-graph formalism, i.e. BIFS.

With respect to the fifth requirement, the SDM specifies each decoder as a separated component, however it does not specify the interface for integrating it. Therefore it is up to the player architecture implementation to design it.

The sixth requirement is not satisfied by the original SDM design.

The next section describes how each of these requirements was addressed.

I.5.2 Optimized SDM design

As it was discussed before, in Section I.3.2.4.1, the MPEG-4 standard proposes an architecture for implementing MPEG-4 decoding applications. However, this architecture is very basic, and does not describe other components necessary for a complete player. Figure 2.10 presents the complete block diagram of the MPEG-4 player.

On the left side of Figure 2.10 *Data Input* blocks can be observed, which represent the components of the player that are responsible for handling different types of input streams. One MPEG-4 scene can receive data from one or more input streams, thus it is possible to have a local MPEG-4 file that receives some scene data, e.g. BBA animation, from a network stream. The communication between the *Data Inputs* and the *Decoders* is performed trough shared data structures (*Input Buffer*) that hold the data sent from the *Data Input* until a *Decoder* is ready to receive it. The data transferred is in a general binary format, hence it does not depend of its type. Therefore it is ensured that *Data inputs* can be integrated without changing the rest of the system, hence the first requirement is met.

The synchronization between different streams is done at the *Data Output* level by using the *Timer* component that holds the time for the current frame. The time is calculated since the beginning of the rendering for the current MPEG-4 scene. The *Timer* starts counting at the time when all streams that have frames that need to be played at time-stamp zero have at least one decoded frame, thus ensuring perfect synchronization. The *Timer* value is frozen at the beginning of each frame, and then it is used to retrieve the corresponding packet from the *Output Buffers* and present it. This allows skipping late frames, thus avoiding synchronization problems.

One solution for solving the requirement to support multiple scene-graph formalisms is to create an intermediate scene-graph format, that will be then used for rendering. As

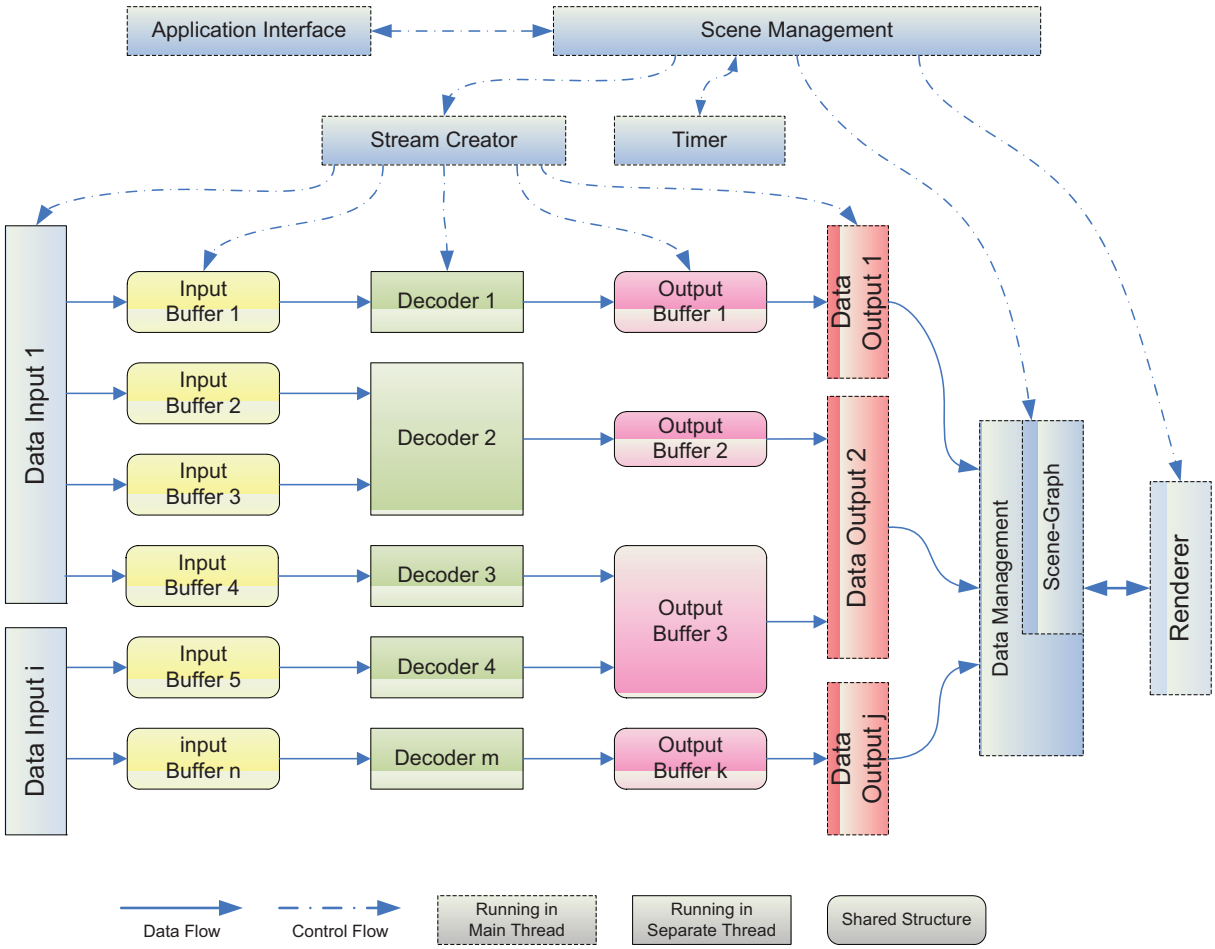


Figure 2.10: Block Diagram of the MPEG-4 Player for PC

it was stated in Section I.4, Table 2.1, it is not necessary to implement all BIFS nodes, hence an intermediate scene-graph is a viable solution. Furthermore, it separates the input scene-graph format from the rendering one, thus allowing better optimization at the *Renderer*.

The intermediate scene-graph is further used to standardize the *Decoder* to *Data Output* communication, hence all decoders that produce mesh data should convert it into that format. Furthermore, other types of decoders, like those for image, video, sound and BBA animation, have predefined output format, hence allowing for easy integration of new ones.

It can be observed from Figure 2.10 that the *Application Interface* communicates only with the *Scene Management* component. The communication is based on a predefined interface, having only the *Application Interface* that depends of the *Scene Management* and not vice-versa. This allows having multiple *Application Interfaces* without changing the rest of the system, therefore solving the requirement for easy integration in more environments.

The last requirement is implementation oriented, but nevertheless it is also important. Nowadays most of the computers include processors that have two or more sub-processors or cores, hence it is important to design an application that will use all cores if possible. The design of the MPEG-4 player takes this into consideration by implementing different tasks in separate threads. The Main thread is executing the core of the application, while additional threads are created when opening MPEG-4 scenes. Each *Decoder* and *Data*

Input is running in its own thread, while they pass data to each other and the main thread by using the shared *Input Buffer* and *Output Buffer* structures. The decision of which thread is executed at which core is left to the operating system. This solution tackles the last requirement.

Table 2.3 summarizes the requirements and how they were solved.

Table 2.3: Requirements for a MPEG-4 Player for PC

Requirement	**Solution**
Receive data from multiple steams, either file or network	Define a standard interface for different input streams
Synchronize all data streams	Implement a common timer component
Accommodate multiple scene-graph formalisms	Define an intermediate scene-graph
Easily integrate different decoders	Define a standard interface for integrating decoders
Easily integrate in different applications	Predefine functions and messages for communicating with the player architecture
Utilize optimally the capacity of the client terminal	Separate different tasks in more threads

In this section it was shown how the different requirements were met by the design the of the player. The next section presents the implementation in more detail.

I.5.3 MPEG-4 Player Components

As it can be observed from Figure 2.10, the player is separated in the following modules:

- Scene Management;
- Application Interface;
- Stream Creator;
- Timer;
- Decoding;
- Data Management;
- Rendering;
- Scene-graph.

Appendix C presents the classes included in the player and their inheritance.
The following sections explain each of the modules in detail.

I.5.3.1 Scene Management

This module is the core of the Player, executed in the main thread, and used to initialize, connect and synchronize the operation of the other modules. Therefore, when the program is executed, this is the only module called from the *Application Interface*. Except for its initial creation, every other event passes trough it.

In order to separate the *Application Interface* from the rest of the player, the *Scene Manager* was designed with clearly defined input and output functions. The data passed to the functions does not depend on the application interface, or on the operating system. When the user executes an action on the *Application Interface*, it calls this module, that in turn forward the action to the appropriate component. An action may be a mouse movement, key press, or a custom action such as the one for opening a file or change rendering settings.

The implementation of *Scene Management* exposes six important methods:

- Init - Initializes the complete architecture.
- Destroy - Deletes all data and destroys the architecture.
- ProcessMessage - Processes a message from the user.
- LoadFile - Loads a file in the architecture.
- CloseFile - Closes a loaded file.
- RenderScene - Renders one frame of the scene .

I.5.3.2 Application Interface

The *Application Interface* depends on the different types of user environments. It creates the user interface, initializes the *Scene Manager* and sends the user input to it. Currently four interfaces are supported:

- The **Windows application** interface is the most versatile. It contains a menu that can be used to see the current status of the rendering options, options to save and load the current window and status to and from an *ini* file. Additionally it has an option to capture a movie of the rendering, as well as an option for command-line loading of scene, and capturing a screen-shot from that scene.
- The **ActiveX interface** can be used in other applications, or attached to a website, while it only exposes a single function for loading a scene.
- The **Mozilla plug-in** interface is a specific for use in Mozilla web browsers. It has the same options as the ActiveX interface.
- The **Chrome extension** interface is a specific for use in the Chrome web browser. It has the same options as the ActiveX interface.

Because of the nature of the interfaces, the player is compiled separately for each one.

I.5.3.3 Stream Creator

The *Stream Creator* module is responsible for creating and managing all data streams that flow from the input to the rendering module, hence it contains information for all *Data Input*, *Decoder*, *Data Output* and *Buffer* structures that are used by a MPEG-4 scene. Since it is responsible for creating the data streams, it also holds information on how each stream type is handled, i.e., which *Decoder* and *Data Output* is used.

Initially the *Stream Creator* creates the appropriate *Data Input* for the input MPEG-4 scene type. After that, for each found stream, the *Data Input* calls the *Stream Creator* to create the relevant *Decoders* and *Data Outputs* and connects them by using the shared buffer structures, hence a decode chain is created. Furthermore, it is used to execute some action on all *Data Inputs* like checking whether all of them have at least one frame available for display.

The implementation of *Stream Creator* exposes four important methods:

- CreateDataInput - Creates a specified data input.
- CreateDecodeChain - Creates a decode chain for a stream.
- AreAllStreamsLoaded - Checks whether all streams have at least one frame available.
- PlayAllStreams - Executes the play function of all active data outputs.

Furthermore, the *Stream Creator* is used to optimally create the decoders, in such a way that the CPU on the terminal does not get overloaded with using too many threads. One optimization is done for decoding images because in an MPEG-4 file the number and size of images, i.e. textures, can be far greater than those of other streams. If it is allowed to create a decoder, i.e. thread, for each image, they can use most of the processing power and do not allow the other types of decoders to decode the data, leading to a big delay in the loading of the scene. Therefore the *Stream Creator* creates one decoder for each image type and all images of same type are forwarded to the same decoder.

I.5.3.4 Timer

This component tracks the time that is measured and its value is read at each rendered frame. The time value is used by different data outputs, so they can compose the correct data.

The *Timer* exposes the current time relatively to the start of the MPEG-4 scene, as well as methods to stop and start the timer. The most important function is the one used to freeze the time value that is executed before each rendered frame. Then this value is used to synchronize the rendering of the animated components, especially in the case when multiple animations are present. Therefore they will all compose data that has the same value for the time-stamp, independent of the decoding time.

The implementation of *Timer* exposes three important methods:

- ClockReset - Resets the timer clock to zero.
- Frame_Lock - Stores the current time.
- Frame_Time_Get_ms - Retrieves the current time.

I.5.3.5 Decoding

The role of the *Decoding* module is to get the data from the input stream, transform it in to a compatible format and send it to the output. It has multiple components (Data Input, Decoder and Data Output) and each of them is responsible for different tasks in the decoding process. The role of each component is:

- Data input - Gets the data from different sources (file, stream, ...), demuxes the data and sends the different types of data to the decoders.
- Decoder - Receives the data from the input, decodes it into a predefined format, and sends it to the output.
- Data Output - Receives the data from the decoder, and according to the type of data it prepares it for composition.

The data between the DataInput and the Decoder, and between the Decoder and the DataOutput is transferred using the Buffer class. This class contains a list of frames (Frame). A frame can be a simple buffer, or a specific frame dependent of the stream type (FrameMesh, FrameAnim, FrameAudio, FrameImage). Each of these frame types can contain additional data, for example the height and the width of the image in FrameImage. Each frame contains information about the stream that the frame belongs to and a time-stamp. The access to the frames in the Buffer from the different threads is controlled by a mutex (to control the access to the frames) and a semaphore (to count the available frames).

The Data Output components play a major role in the composition process and are responsible for transferring the decoded data from memory to the graphics card, making a connection between the scene-graph and the different resources (textures, animation, etc.) and requesting decoding of new resources as well as new data inputs (e.g. streaming of BBA).

The *Data Input* interface defines only three functions that need to be implemented by a specific data input:

- Init Initialization of the data input, and creates the streams.
- GetTrackData Returns one frame of the requested stream.
- Destroy Frees memory and closes the data input.

The *Decoder* interface defines only three functions that need to be implemented by a decoder in order to be included in the architecture:

- InitDecoder Initializes the decoder by using the first data that is sent from the data input. This can be a decoder specific info if it is available, or the first data frame.
- Decode Decodes a frame from the input data and creates an output frame that is returned from the method.
- FreeDecoder Frees memory and closes the decoder.

The *(*Data Output) interface defines only three functions that need to be implemented by a specific data output:

- Init Initializes the data output.
- Play Prepares the decoded data and sends it to the specific output.
- Destroy Frees memory and closes the data output.

As it can be observed from the implementation, the interfaces are quite simple, enabling easy integration if other decoders and input streams are needed.

I.5.3.6 Data Management

This component is used to store information about all created data, including 3D meshes, text and texture information. Depending of the selected options, the needed data is grouped and sent to the *Renderer* for display. Furthermore, it controls referencing of resources. For example if an image is requested and has been already decoded, its reference will be returned.

The *Data Management* component exposes the following important methods:

- AddMeshGroup - Adds a new mesh to the mesh registry.
- AddDeformer - Adds a new deformer to the deformer registry.
- AddText - Adds a text to the texts registry.
- LoadImage - Requests decoding for an image used as texture.
- Render - Calls the Render method from renderer with the selected objects.

I.5.3.7 Rendering

The *Rendering* module is used to display the scene on the screen. In order to allow interoperability with different operating systems, it is necessary to separate the rendering part from other components. Therefore the *Rendering* module defines an interface for integrating different rendering APIs or graphics engines and includes the following components:

- Renderer - Interface for the main rendering engine.
- Mesh - Interface for storing mesh data.
- Texture - Interface for storing texture data.
- Deformer - Interface for storing deformers that are used for bone-based animation and morphing.

The *Renderer* interface exposes the following important methods:

- Init - Initializes the rendering engine.
- Cleanup - Destroys the rendering engine and all that was created by it.
- CreateMeshGroup - Creates a mesh from the input mesh data.
- CreateTexture - Creates a texture from the input pixel data.

- CreateDeformer - Creates a specific deformer for a mesh.
- CreateText - Creates a text overlay from the input string.
- AddMeshGroup - Adds a mesh in the rendering pipeline.
- BeginRender - Initializes the rendering process.
- Render - Renders one frame.
- EndRender - Draws the text overlay and finishes rendering.
- ProcessMouseMessage - Handles mouse input from the user for moving the camera.

As it can be observed from the methods, the *Renderer* component is used to create instances of the other components, which is a necessity because each graphics API has its own data structures.

I.5.3.8 Scene-Graph

The *Scene-Graph* module is defining a scene-graph structure that is filled with the decoded data from the MPEG-4 scene, hence it separates the MPEG-4 decoders specific structures from the other components of the player. Furthermore it allows for different optimizations to be made on the scene-graph data, independent of the input scene-graph format.

One optimization is done when transferring mesh information from the original scene-graph format to the player scene-graph format. How mesh is stored in the input scene-graph is different in most scene-graph formalisms. For example, in BIFS, each component (coordinates, normals, texture coordinates) and its indices are stored in separate arrays. However, in COLLADA the indices are stored multiplexed in one single array. On the other hand, graphics cards usually take for input array of vertex information and only one index list, hence all data needs to be transformed into a unique graphics card format. In the case of COLLADA, the indices first have to be demultiplexed and then the mesh data can be converted. For BIFS, the data is converted directly. The conversion is done using the following algorithm:

1. Create new arrays for vertices and all parameters (normals, texture coordinates, etc.) that exist in the original mesh and create new index arrays.
2. For each index group (containing the indices in each of the index arrays having the same position) check if that group of values existed previously in the index arrays.
    a) If yes, put a new index into the new index array that points to the same vertex as the previously found group.
    b) If no, add a new vertex and parameters having the values that the indices point to, and add a new index that points to this vertex.

From this data, a new node is created and it is used for rendering.

Another optimization is done with respect to the animation, more specifically the bone based animation. The animation data is composed of two parts: bone hierarchy and bone influences. This data is stored differently in both BIFS and COLLADA. In BIFS, the hierarchy is made out of SBBone nodes that contain the initial transformation of the bone with respect to the parent, as well as information for the bone influences on the vertices,

composed of pairs of vertex indices and weights. On the other hand, in COLLADA, the hierarchy is made of JOINT nodes and the top element points to a controller node that contains the initial transformation of the bones, as well as the bones influences. The influences are stored in three arrays: the first array stores all used weight values, the second array stores the number of influences per vertex and the third array stores pairs of bone and weight indices for each influence for each bone.

As it can be observed both representations are different. Furthermore, there are some limitations imposed by the animation processing on the graphics card: (1) an inverse world bind pose matrix has to be calculated for each bone, (2) it does not optimally handle a hierarchy of bones and (3) all bone influences have to be stored per vertex. Therefore, in the player scene-graph, a unified format close to the graphics card is used containing a list of matrices for each bone, as well as list of (bone index, bone weight) pairs for each vertex of the animated mesh.

The processing is done in two stages: (1) calculation of the inverse world bind pose matrix and (2) creation of the vertex influence list. For BIFS, the hierarchy is passed and the matrix is calculated for each bone, and at the same time the influences of that bone are added to the vertex influence structure. For COLLADA, first the hierarchy is passed and the matrix is calculated for each bone, and then the controller node is parsed, and the bone influences are transferred to the vertex influence structure.

The third optimization is done with respect to the morphing animation, which is composed of two data structures: a list of vertex arrays corresponding to morph targets and a list of weights used to calculate the final shape. The representation is similar in both BIFS and COLLADA, however with the optimization of the vertices array of the base shape, the target shapes no longer have the same array length. There are two solution for the processing of the morph shape: (1) create a mapping between the new vertex position and the old one and (2) expand also the target shapes when expanding the base shape. The first solution is more memory efficient, however it requires that the vertex data be taken from the graphics card and updated with the new positions, an operation that can be very costly. The second solution requires more memory, however the calculation of the new position can be done directly on the graphics card by using vertex shaders (there is a limitation on the number of target shapes that can be used, depending on the graphic card).

As it can be observed from the design process of the MPEG-4 player, accessing MPEG-4 content can be very complex. In order to facilitate the process, a new middleware was designed and it will be presented in the next section.

I.6 MPEG Extensible Middleware (MXM)

As it can be observed from the previous section, accessing MPEG-4 content requires deep knowledge of the MPEG-4 standard. Although there are already frameworks for accessing MPEG-4 content including the reference software and third party implementations like GPAC, development of MPEG-4 capable software is not facilitated enough. Therefore, having a simplified API can be of a great importance for spreading the usage of the standard.

The role of the MPEG Extensible Middleware (MXM) is to tackle exactly that problem. We contributed on the standardization process of MXM on the 3D graphics aspects. MXM will enable the development of a global market of:

- MXM applications that can run on MXM devices thanks to the existence of a standard MXM application API and MXM devices executing MXM applications thanks to the existence of a standard MXM architecture.
- MXM engines thanks to the existence of standard MXM architecture and standard APIs.
- Innovative business models because of the ease to design and implement media-handling value chains whose devices interoperate because they are all based on the same set of technologies, especially MPEG technologies.

MXM specifies a set of Application Programming Interfaces (APIs) so that MXM applications executing on an MXM device can access the standard multimedia technologies contained in its Middleware as MXM engines.

The APIs belong to two classes:

- The MXM engine APIs, i.e. the collection of the individual MXM engine APIs providing access to a single MPEG technology (e.g. video coding) or to a group of MPEG technologies where this is convenient.
- The MXM orchestrator API, i.e. the API of the special MXM engine that is capable of creating chains of MXM engines to execute a high-level application call such as Play, as opposed to the typically low-level MXM engine API calls.

The following MXM Engine APIs are currently implemented:

- Digital Item - defines interface for operating on ISO/IEC 21000-2 Digital Item Declaration (DID) data structures.
- MPEG-21 File - defines the methods for operating over ISO/IEC 21000-9 MPEG-21 File Format files.
- REL - defines the methods for operating over ISO/IEC 21000-5 Rights Expression Language (REL) data structures.
- IPMP - defines the methods for operating over ISO/IEC 21000-4 Intellectual Property Management and Protection data structures.
- Media Framework - defines grouping together several media specific engines such as Video, Image, Audio and Graphics Engines. It also implements common functionalities (independent on the media type) such as resource loading and saving.
- Meta-data - defines the methods for operating over meta-data structures.
- Digital Item Streaming - defines the methods for operating over ISO/IEC 21000-18 Digital Item Streaming data structures.
- Digital Item Adaptation - specifies means to access and create information pertaining to the usage environment context where Digital Items or media resources are ultimately processed, consumed, created, distributed, etc.
- Event Reporting - defines the methods for operating over ISO/IEC 21000-15 Event Reporting data structures.

- Content Protocol - defines the methods for for performing Content Protocols as specified in ISO/IEC 29116-1.
- License Protocol - defines the methods for for performing License Protocols as specified in ISO/IEC 29116-1.
- IPMP Tool Protocol - defines the methods for for performing IPMP Tool Protocols as specified in ISO/IEC 29116-1.
- Content Search - defines the methods for searching for content.
- Security - defines security-related methods.
- MVCO - lets an MXM Application access the functionalities of the Media Value Chain Ontology specified in MPEG-21 Part 19. Media Value Chain Ontology (Committee Draft)
- Domain - defines methods for operating on ISO/IEC 29116-1 Domain management data structures.
- Rendering - defines a number of interfaces allowing rendering of a scene.

Since accessing content is the main interest point, the Media Framework Engine will be described in the next section.

I.6.1 Media Framework Engine

This section presents the work of the author that was done for and included in the MXM API. The main goal of the work is to propose a solution for converting the MPEG-4 content into a more comprehensible and easily consumable format.

Figure 2.11 presents the classes included in the Media Framework Engine. Two main groups can be observed: the first one, Access API, is used to load content from MPEG-4 files and the second one, Creation API, is used to create MPEG-4 files.

Both groups have the same type of APIs. The *Engine* is the main API that is used to open MPEG-4 files and gives access the other APIs, which are used for a specific type of content. Four main APIs exist:

- Audio - API for dealing with audio content.
- Image - API for dealing with image content.
- Video - API for dealing with video content.
- Graphics3D - API for dealing with Graphics3D content.

All of these APIs are simple and expose only few methods to access the data, however they can still cover a wide range of applications. An application that needs to access only video and audio streams, needs to use only a few methods to get the data.

While the data from audio, video and image is a simple one, the content from a graphics scene is more complex and versatile. Therefore, the Graphics3D API is more complex, however it is still much simpler than accessing the content in the standard way. The next section describes the Graphics3D Access API in more details. Let us note that this API was proposed by the author to MPEG for standardization in the process of MXM.

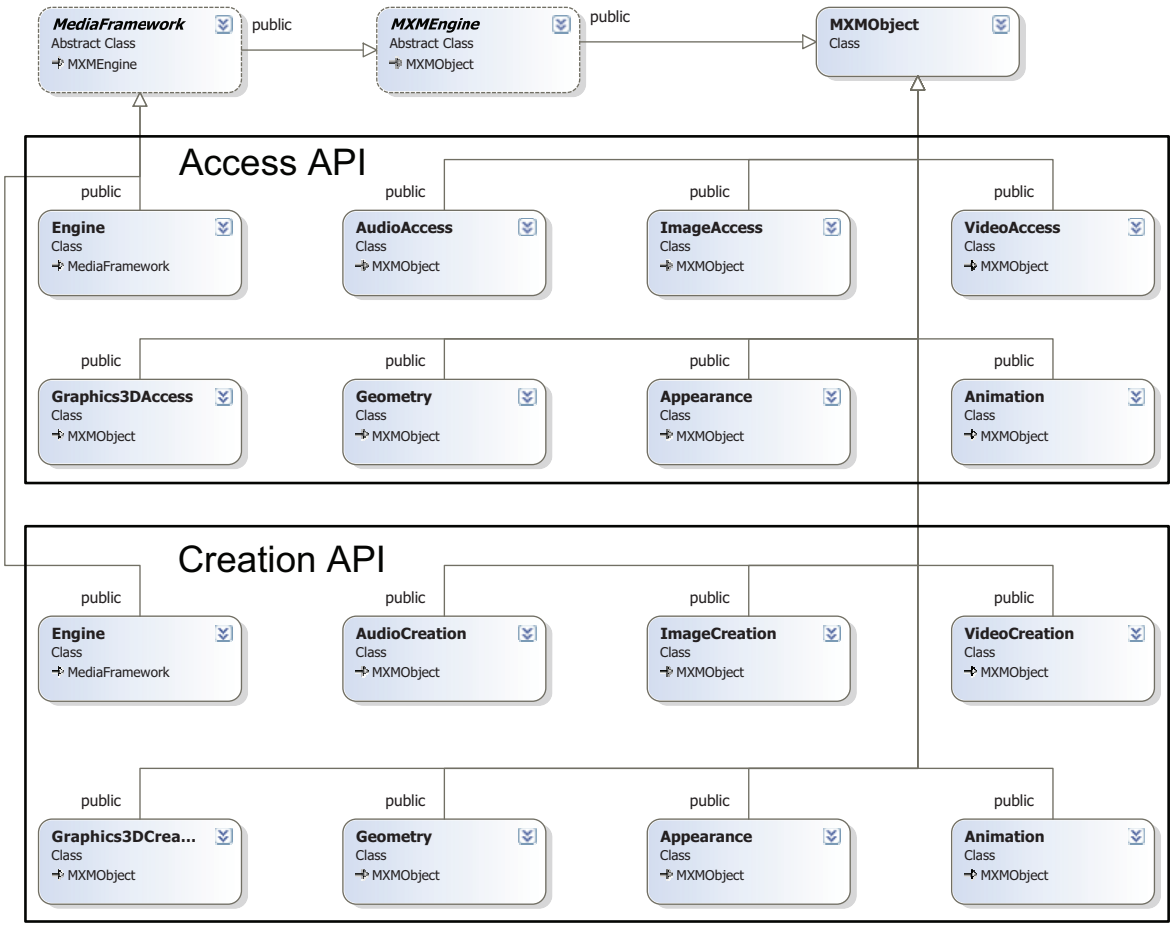


Figure 2.11: Class diagram of the Media Framework Engine

I.6.1.1 Graphics3D Access API

The Graphics3D API gives access to the 3D content in the scene of the MPEG-4 file. It parses the scene and converts each mesh into a flat buffer representation that is simple enough for easy parsing by an external application.

The API is separated in three parts, each of them specialized for different part of the content:

- Appearance - API for dealing with mesh appearance.
- Geometry - API for dealing with 3D mesh shape.
- Animation - API for dealing with mesh animation.

The following sections present each of the APIs in detail.

I.6.1.1.1 Appearance API

The Appearance API gives access to the visual appearance of the mesh, including its diffuse, specular, etc. colors and a reference to a texture. The main function of the API is **GetAppBuffer** that returns a buffer containing a description of the appearance for each mesh of the scene.

Table 2.4: Appearance Buffer

```
Atom AppearanceData
{
  Int(4) sizeAppearanceData;
  Int(1) appearanceCount;
  for ( 1 .. appearanceCount)
    Appearance();
}

Atom Color
{
  Float(4) red;
  Float(4) green;
  Float(4) blue;
}
```

```
Atom Appearance
{
  Int(4) textureID;
  Int(1) haveMaterial;
  If (haveMaterial = 1 )
  {
    Color diffuseColor;
    Color specularColor;
    Color emissiveColor;
    Float(4) transparency;
    Float(4) ambientIntensity;
    Float(4) shininess;
  }
}
```

The description of the appearance buffer is presented in Table 2.4.

It can be observed that the structure of the buffer is quite simple, thus parsing it is easy.

I.6.1.1.2 Geometry API

The Geometry API gives access to all geometry in an MPEG-4 file. The geometry may be included directly in the BIFS scene, or compressed by some other technique and referenced in the BIFS scene. The main function to access the geometry is **GetVBandIB**, which returns a buffer that is a flat representation of all geometry.

The description of the geometry buffer is presented in Table 2.5.

It can be observed that the buffer format is relatively simple, and using it does not require knowledge of the internal format of MPEG-4. It organizes the meshes in an array of vertex buffers that contain more index buffers that have separate appearance. The data can be returned in two formats: vertex buffer format, in which the data is flattened and there is only a coordinate index buffer, and a second format, IndexedFaceSet, that keeps the original format of the mesh as it was stored in the scene.

This allows using MPEG-4 files in an application with only writing a few lines of code. However, the simplification of the scene into a flat buffer has a price, it removes some capabilities and features from the standard like access to the scene-graph, scripts and all other advanced features. Since the API is intended to enable easy inclusion of MPEG-4 content in an application, and furthermore considering that most applications implement their own scene-graph and scripting environments, these features can be easily excluded.

I.6.1.1.3 Animation API

The Animation API gives access to an animation, if existent, connected to a mesh. It extends the usage of the APi for applications that require animated 3D graphics content. The API supports two types of animations, bone-based provided by the BBA stream and coordinate interpolation provided by the FAMC stream, hence it has two main methods.

Table 2.5: Geometry Buffer

```
Atom MeshData
{
  Int(4) sizeMeshData;
  Int(1) vbCount;
  for ( 1 .. vbCount )
    VertexBufferDesc();
}

Atom VertexBufferDesc
{
  Int(4) sizeVertexBufferDesc;
  VertexBuffer();
  Int(1) ibCount;
  For ( 1 .. ibCount )
    IndexBuffer();
}

Atom IndexBuffer
{
  Int(4) sizeIndexBuffer;
  Int(4) appearanceID;
  Int(4) coordIndexCount;
  For ( 1 .. coordIndexCount )
  {
    Int(4) index;
  }
  Int(4) texCoordIndexCount;
  For ( 1 .. texCoordIndexCount)
  {
    Int(4) index;
  }
  Int(4) normalIndexCount;
  For ( 1 .. normalIndexCount)
  {
    Int(4) index;
  }
}
```

```
Atom VertexBuffer
{
  Int(4) sizeVertexBuffer;
  Int(1) description;
  Int(4) vertexCount;
  For ( 1 .. vertexCount )
  {
    Float(4) x;
    Float(4) y;
    Float(4) z;
    If ( boneWeights )
    {
      Float(4) w0;
      Float(4) w1;
      Float(4) w2;
      Float(4) w3;
      Int(1) boneIndex0;
      Int(1) boneIndex1;
      Int(1) boneIndex2;
      Int(1) boneIndex3;
    }
  }
  Int(4) normalCount;
  For ( 1 .. normalCount)
  {
    Float(4) nx;
    Float(4) ny;
    Float(4) nz;
  }
  Int(4) texCoordCount;
  For ( 1 .. texCoordCount)
  {
    Float(4) u;
    Float(4) v;
  }
}
```

Table 2.6: BBA Animation Buffer

```
Atom AnimationBuffer
{
  Int(4) sizeAnimationBuffer;
  Int(1)  hasHierarchy;
  If (hasHierarchy)
    Hierarchy h;
  Int(4)  NoOfFrames;
  For ( i = 1 .. NoOfFrames )
    Frame f[i];
}

Atom Hierarchy
{
  Int(4) sizeHierarchy;
  Int(4) charactersCount;
  For ( i = 1 .. charactersCount )
    Bone rootBone[i];
}
```

```
Atom Bone
{
  Int(4) idBone ;
  Int(4) childBonesCount;
  For ( i = 1 .. childBonesCount )
    Bone Bone[i];
}

Atom Frame
{
  Int(4) size;
  Int(4) boneDataCount;
  For ( i = 1 .. boneDataCountt )
    BoneData bd[i];
}

Atom BoneData
{
  Int(1) boneDataMask;
  For each (component in boneDataMask)
    Float(4)  component;
}
```

The first one is *GetDecodedBBA*, which returns a buffer containing all decoded BBA frames. The description of the BBA buffer is presented in Table 2.6.

As it can be observe the structure is quite simple. First it gives the hierarchy of the bones and then the transformation data for each frame.

The second one is *GetDecodedFAMC*, which returns a buffer containing all decoded FAMC frames. The description of the FAMC buffer is presented in Table 2.7.

As the previous ones, this structure is also simple. First it gives the number of frames, and then for each frame its time-stamp and an array of coordinates, normals and texture coordinates.

Mobile device are far less powerful than personal computers, hence the same player architecture cannot be used. Therefore, in order to be able to play MPEG-4 content on a mobile device, many optimizations need to be performed. The next section presents design of a MPEG-4 player for a mobile device.

I.7 Design of an MPEG-4 Player Architecture for Mobile Devices

Several implementations of MPEG-4 graphics are made available in products such as the ones proposed by iVast or Envivio, or in open source packages such as GPAC [33]; however, the literature on MPEG-4 3D graphics on mobile phone is almost inexistent. In order to quantify its capabilities for representing graphics assets as used in games, a MPEG-4 3D graphics player was designed and implemented for the Nokia S60 platform based on the Symbian S60 FP1 SDK [30]. The testing hardware included Nokia N93, Nokia N95 and

Table 2.7: FAMC Animation Buffer

```
Atom FAMCBufer
{
  Int(4) sizeFAMCBuffer;
  Int(4)  NoOfFrames;
  For ( i = 1 .. NoOfFrames )
    Frame f[i];
}
```

```
Atom Frame
{
  Int(4) size;
  Int(4) timeStamp;
  Int(4) coordCount;
  For ( 1 .. coordCount )
  {
    Float(4) X;
    Float(4) Y;
    Float(4) Z;
  }
  Int(4) normalCount;
  For ( 1 .. normalCount )
  {
    Float(4) X;
    Float(4) Y;
    Float(4) Z;
  }
  Int(4) texCoordCount;
  For ( 1 .. texCoordCount )
  {
    Float(4) U;
    Float(4) V;
  }
}
```

Nokia N95 8GB (which have nearly the same hardware and performances). To ensure good performances, the player was implemented in C and C++. For MPEG-4 de-multiplexing and BIFS decoding, the implementation is full software (based on the GPAC framework). The implementation of the rendering is hardware supported (based on OpenGL ES [63]).

I.7.1 Requirements

To properly design the mobile MPEG-4 player, an analysis of the mobile devices needs to be performed and a set of requirements defined.

Because the mobile phones are naturally weaker devices than a desktop or a laptop computer, the architecture of the player should be simpler. Mainly, processors on mobile phones have only one sub-processor, hence parallelizing different tasks in different threads would not bring any benefit. Therefore, decoding the MPEG-4 file should be done sequentially as different streams are needed or requested.

Another problem may arise with the decoding algorithms since most of them are designed for PC. Mobile phones have restrictions in the processing power, as well as memory limitations. Therefore the algorithms should be adapted for mobile phones. Furthermore, implementing all MPEG-4 algorithms on a mobile phone is a tedious task, and it may not be necessary. Therefore, a subset of decoding algorithms should be chosen, having in mind their processing power and memory requirements.

As it was mentioned before, when the requirements for a PC MPEG-4 player were

discussed in Section I.5.1, with the introduction of MPEG-4 Part-25, different XML scene-graph formalisms can be used inside MPEG-4 files. However, implementing them will add significant complexity to the player and increase its memory requirements. Therefore, the player will include support only for the standard MPEG-4 scene-graph format, namely BIFS.

Connectedness to the Internet is one of the most important features of mobile phones, hence it is necessary to implement an player architecture that can use it to access data. Therefore, the player architecture should be capable of receiving data from multiple streams, hence it opens opportunities for different applications.

The following main requirements can be derived:

- Simplify the architecture.
- Find decoders appropriate for mobile devices.
- Reduce the number of BIFS nodes.
- Support input from multiple streams.

The next section introduces the architecture of the player and presents how the different requirements were met.

I.7.2 Implementation

Figure 2.12 represents the block diagram of the architecture of the mobile player.

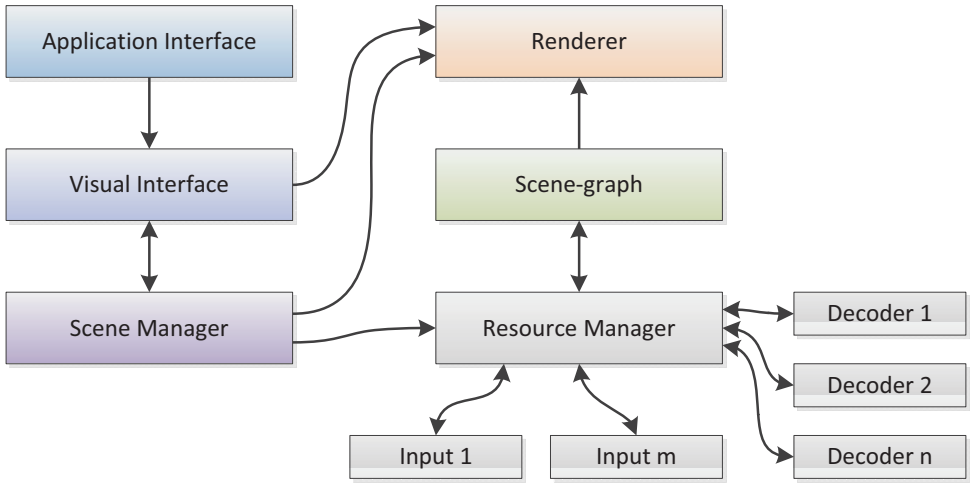


Figure 2.12: Mobile MPEG-4 player architecture

As it can be observed from Figure 2.12, the player architecture is simpler than the one designed for the PC player presented in Section I.5.2. Every task executes in the same thread and in the order that is requested. For example, when the user requests to open a file, the *Visual Interface* sends this message to the *Scene Manager* that forward the request to the *Resource Manager*. Then it opens the file with the appropriate *Input*, and decodes the BIFS scene with the BIFS decoder. While the scene is parsed, every resource (e.g. texture) is requested from the *Resource Manager* that, if the resource was not decoded, decodes it using a *Decoder* and then it returns it. When the scene is ready, the *Renderer* displays it.

In order to be able to decode the data as fast as possible, the coding algorithms should be chosen carefully. The most important data is the mesh description. There are several algorithms that can compress it effectively, however the complexity of the decoder is very high for a mobile phone. Therefore only the BIFS scene-graph compression is used to represent the mesh data. For representing the animation, the BBA codec is used, however it was optimized for running on a mobile phone [55]. The textures are also important assets, however their size is significant compared to the other ones. Therefore the compression ratio is more important than the complexity of the codec, hence JPEG2000 compression is used.

BIFS is very complex, containing many nodes that are not necessary for the targeted mobile applications, hence a subset of nodes should be chosen to be used for simplifying the implementation. This subset of nodes was presented previously in Section I.3.2.5.

As it can be can observed from Figure 2.12, the player architecture is separated in several modules:

- Application Interface;
- Visual Interface;
- Scene Manager;
- Renderer;
- Scene-graph;
- Resource Manager.

I.7.2.1 Application Interface

The Application Interface module is dependent of the operating system. It is used to initialize the applications and create all other modules. The main function is to create the application window, however all user interface is rendered independently of the OS.

I.7.2.2 Visual Interface

The Visual Interface is a module that uses the Renderer to display the interface of the application to the user. It has support for displaying buttons which can be activated by touch events, as well as text and file browsing. The interface is adapted automatically to the resolution and orientation of the screen of the device, hence it is usable for different mobile phones.

I.7.2.3 Scene Manager

The Scene Manager is responsible for managing all other components. It request loading of a file, update of animation and calls the rendering of the scene-graph. Furthermore, it manages all states of the player, and switching between them on user actions. The states of the player include: main screen, file browser screen, scene rendering screen and help screen.

I.7.2.4 Renderer

The Renderer is responsible for all display actions in the player. It has methods for displaying different kinds of data. When the program is started, the Renderer initializes the screen and loads the fonts. When a scene is loaded, the methods of the Renderer are called to display the data. Furthermore, it is used to display the user interface, hence the connection from the Visual Interface to the Renderer.

I.7.2.5 Scene-graph

The Scene-graph is used to represent the scene that is to be rendered and it is an implementation of the BIFS scene-graph format. It does not support all BIFS features, however it supports updates of the scene, which are important for game applications. The rendering code for each node is defined in its rendering method, hence all nodes render themselves. The rendering is called just for the top node that propagates it trough the hierarchy.

I.7.2.6 Resource Manager

The Resource Manager is very important for keeping track of all resources loaded for one MPEG-4 file. Resources include MPEG-4 files, textures and animation. This component provides easy access to the resources, and makes sure that each resource is decoded and loaded only once. Furthermore, when all references to a resource are released, that resource is unloaded.

I.8 Conclusion

This chapter presented the main contribution of this dissertation.

The first section proposed a theoretical framework for describing distributed architectures for rendering 2D and 3D graphics. The following basic transformations were defined: rendering, coding, simplification, modeling and scene updates. This framework enables to mathematically define the distributed architectures. By using the previously defined formal framework functions, the architectures that were presented in the state of the art were mathematically modeled. The functions were interconnected and reordered in order to model each architecture. This allowed to analyze the state of the art architectures on a common ground and concluded that none of them is appropriate for distributed game applications.

Because none of the existent distributed architectures is able to answer to the requirements of games on mobile devices, the proposed framework was used to define a new architecture. This architecture is a distributed one, considering the game logic on the server side, and only the rendering one the client. All components of the architecture were analyzed, and a solution for each of them was proposed. Furthermore, the architecture uses the MPEG-4 standard to facilitate the development of games. In the proposed distributed architecture, a key component is the 3D rendering performed on the client. Therefore, the second part of this chapter proposed an optimized approach for rendering MPEG-4 3D content, both on a powerful platform (PC) and on less powerful mobile platform. For the design of the MPEG-4 player for PC, six requirements were defined, however only few of them were addressed previously by the traditional SDM. A new player architecture is proposed that addresses all of the requirements and the proposed implementation was described. By observing the complexity of the MPEG-4 player architecture, it can be concluded that the access of MPEG-4 data is difficult. Therefore a new framework for accessing MPEG-4 content was proposed, with the objective to facilitate the interface. Each of the components of the framework was presented, as well as all of the buffer formats for each type of data (3D mesh, animation, etc.). The last section of this chapter presented the design of the architecture for an MPEG-4 player for mobile devices. Four main requirements were defined, specifying which parts of the PC-based architecture need to be optimized in order to apply it to less powerful mobile devices. Each of these requirements was analyzed and a solution was proposed. The next chapter intends to validate newly the proposed distributed system architecture. Furthermore, it will present the validation of the MPEG-4 players for PC and mobile devices and the proposed simple MPEG-4 access framework.

Chapter 3

Experiments and Validation

Abstract

This chapter presents the validation of the distributed architecture proposed in the previous chapter as well as experiments and validation for the MPEG-4 player architectures. Then the MPEG-4 player architecture for a mobile device is presented and experiments are performed for determining the maximum complexity of 3D content that can be supported by the used mobile device. In order to validate the proposed distributed architecture, a game was adapted to use it. By using this game, experiments were performed in order to determine the class of games that can use this architecture as well as their complexity.

The second part of the chapter presents the MPEG-4 player architecture for PC is validated by demonstrating how different MPEG-4 content samples are loaded and by implementing an on-line animation system for cued speech language. The proposed MXM API is then validated by integrating an MPEG-4 content loader in a third party engine.

I.1 Alternative Client-Server Architecture for 3D Graphics on Mobile Devices

Section I.4 proposed an alternative distributed architecture for supporting games on mobile devices. Section I.7 proposed an architecture for an MPEG-4 mobile player that is to be used as a client in the proposed architecture architecture. Because of different constraints in terms of client capacity, network bandwidth and latency, experiments were performed in order to obtain the boundaries of each of these parameters.

I.1.1 MPEG-4 Player for Mobile Devices

This section presents the client components of the distributed system architecture presented in the previous chapter: the Rendering component (**R**) and a part of the Coding component (**C**). As illustrated on Figure 2.7, the coding component is separated between the server and the client. On the client, the transmitted data is first decoded and then rendered. Therefore one of the objectives of our experiments is to find out the upper limits in terms of 3D assets complexity (with respect to geometry, texture and animation) while ensuring fast decoding (it should be transparent to the player that the asset was first decoded before rendering). The second objective is to analyze the rendering performances of the selected platform.

The first tests performed addressed MPEG-4 files containing only static and textured objects with different number of vertices and triangles. The BIFS decoding time was measured, as well as the rendering frame-rate for each file. The second test is related to animated objects based on skeleton-driven deformation approach. This kind of content, requiring more computations due to the operation per vertex performed during the animation, leads to lower rendering frame-rate than the static objects.

Figure 3.1 and Figure 3.2 illustrate the classical behavior of decoding and rendering capabilities against the number of low-level primitives (vertices, normals, triangles...) for static and animated objects, respectively. The vertical axis on the left represents the BIFS decoding time in milliseconds, and the vertical axis on the right represents the frame-rate in frames per second (fps). Let us note that considering only vertices on the horizontal axis does not provide a complete analysis for BIFS decoding, since the structure of the object graph is not flat as in the case of the structure used when the object is rendered. Indeed, when defining an object in BIFS it is possible to create different index lists for each primitive (vertex, normals, texture coordinates, colors ...); hence the total number of low-level primitives is reported on the horizontal axis.

In order to reduce the decoding time, one solution is to find a more compact manner of representing the low level primitives. The number of index lists is reduced by pre-processing the object in order to flatten its structure: vertices, normals and texture coordinates are sharing a common list. This pre-processing has an impact on the length of each individual primitives' arrays (slightly increasing them), but overall it reduces the size of the compressed data and implicitly the decoding time (around 15%). Furthermore, it does not change the appearance of the object, just the representation of its 3D data. Figure 3.3 shows the comparative BIFS decoding time obtained with and without pre-processing.

Concerning the capabilities of MPEG-4 for decoding and N95 for rendering, it can be observed that: to obtain acceptable decoding time (less than 2 seconds), a 3D asset represented in MPEG-4 should have less than 58 000 low level primitives (corresponding

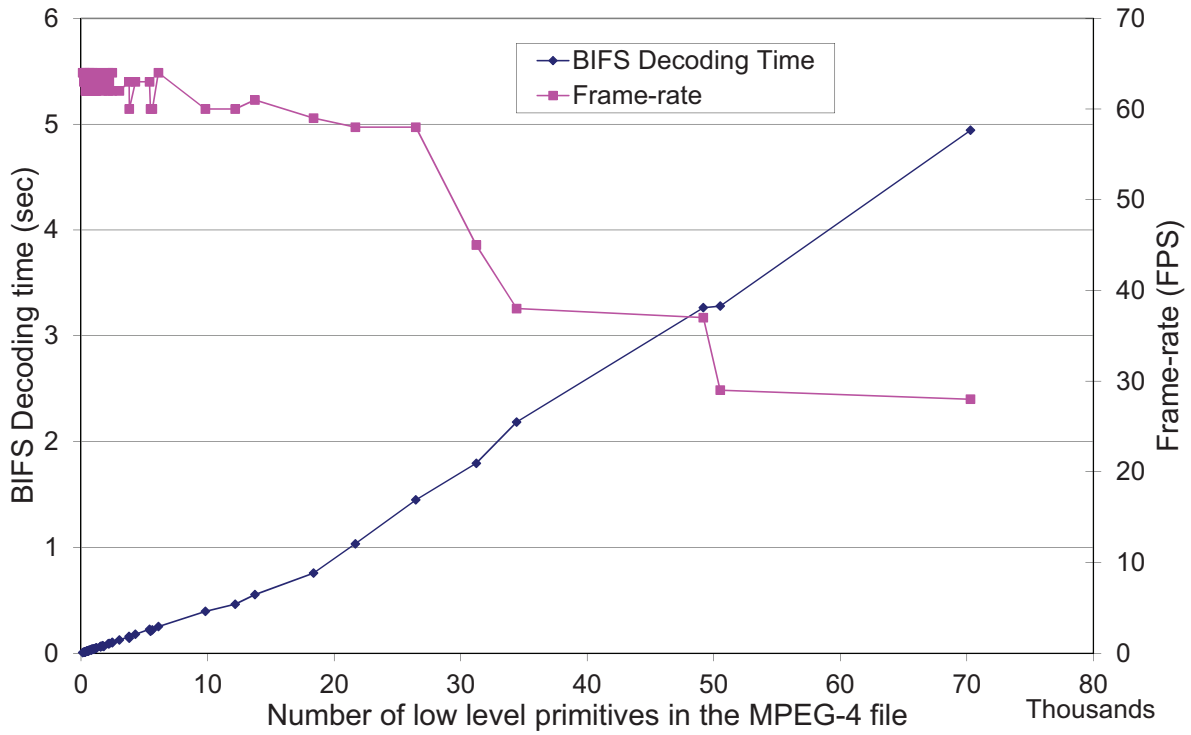


Figure 3.1: Decoding time (in s on left side) and rendering frame-rate (in fps on right side) for static objects with respect to the total number of low level primitives in the MPEG-4 file

to objects with less than 12 000 triangles) for static objects and 1 500 triangles for animated objects. N95 is able to render textured and lighted static objects of around 20 000 triangles at an acceptable frame rate (25 fps) and it is able to render at the same frame rate textured and lighted animated objects of around 6 000 triangles. Figure 3.4 presents snapshots of the player loading different static and animated objects used for tests.

The reduced display size of the mobile phones implies that only few pixels are used to render (sometimes dense) meshes. Simplification for geometry and texture can be used without affecting the visual perception of the objects. By doing so, the size of the content can be reduced by 60-80% and the loading time by 70-90% as it was researched by Preda et al. in [56]. The size of the geometry is reduced by Garland's quadric error metrics technique [35], and the size of the texture by reducing its width and height. Figure 3.6 shows the original Hero model and its version simplified at 27%. The size of the simplified file is 428kB (24% with respect to the original one) and the total loading time is 3.8 sec (57% faster than the original one).

The two tests performed on a large database (around 5 000 graphics files of different nature) indicate that MPEG-4 may offer appropriate representation solutions for simple static and animated objects, both in terms of loading time and rendering frame-rate, by offering a good compromise between compression performances and complexity.

I.1.2 Architecture Validation by a Game

To validate the architecture, this section presents experiments run with a simple car racing game. To quantify the impact of network latency, a game was implemented and tested

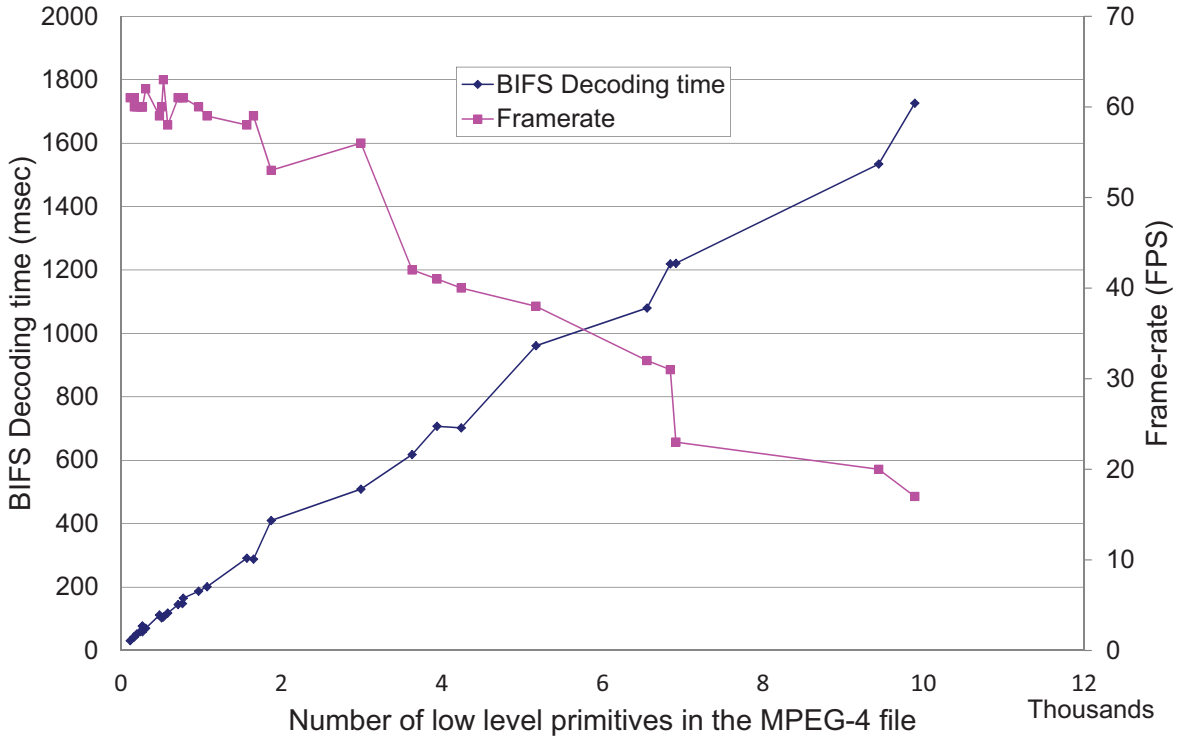


Figure 3.2: Decoding time (in ms on left side) and rendering frame-rate (in fps on right side) for animated objects with respect to the number of low level primitives in the MPEG-4 file

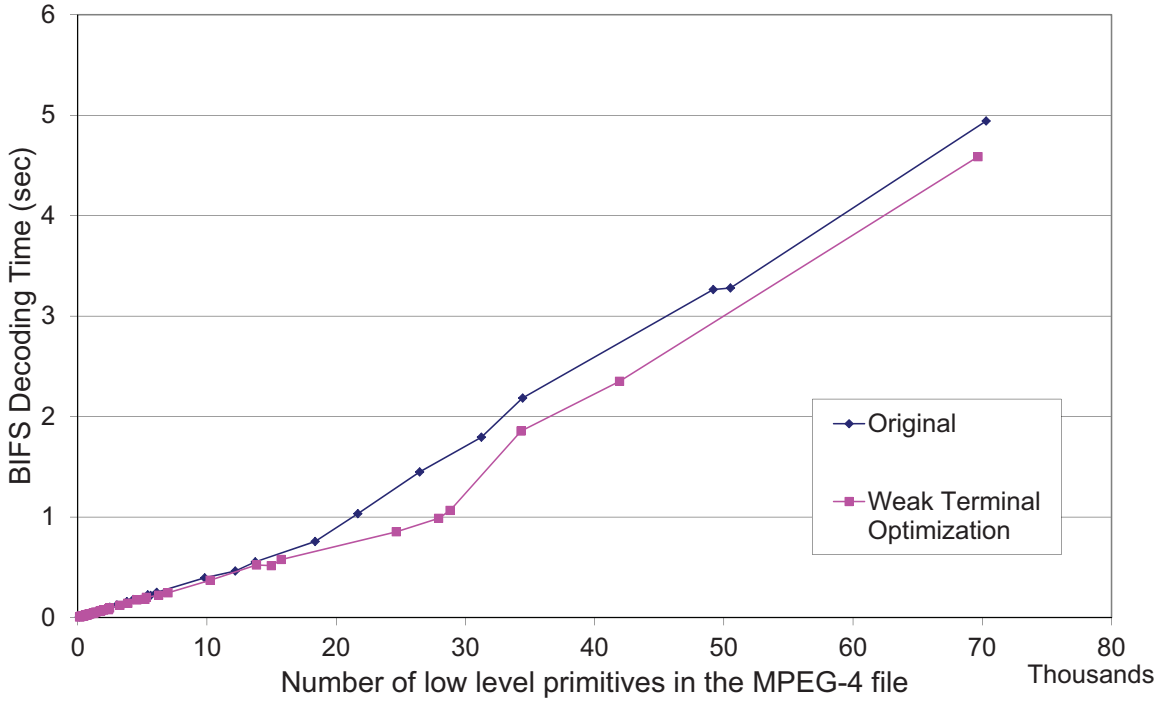


Figure 3.3: Decoding time improvement (in msec) due to the 3D assets pre-processing. On the horizontal the total number of low level primitives

(a) decoding time 1.6 s, rendering frame-rate 63 fps

(b) decoding time 2.2 s, rendering frame-rate 59 fps

(c) decoding time 2.2 s, rendering frame-rate 62 fps

(d) decoding time 1 s, rendering frame-rate 61 fps

Figure 3.4: Snapshots for static (a and b) and animated (c and d) 3D graphics objects

(a) Original model, 6846 triangles, decoding time 8.8 s, rendering frame-rate 32 fps

(b) Simplified model, 1874 triangles, decoding time 3.8 s, rendering frame-rate 58 fps

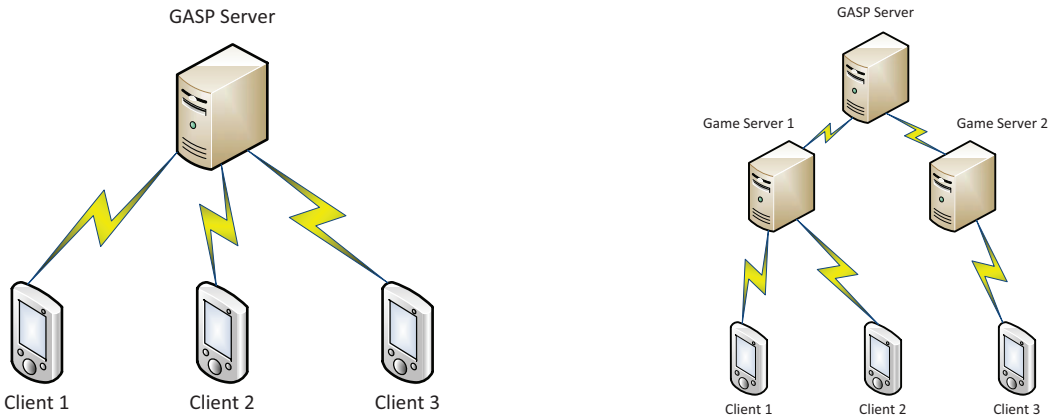
Figure 3.5: The original Hero (a) and its simplified version (b)

them in different conditions. Furthermore, a methodology of 6 steps was defined to adapt an existing game to run in the proposed architecture.

I.1.2.1 Car Racing Game

Based on the architecture proposed in the previous section, a multi-player car racing game was implemented. The game state changes frequently, making it appropriate for testing the architecture. The game was originally developed in J2ME as a traditional multiplayer game for mobile phones. The users control only the speed of the car, however there are two additional parameters which effect its maximal speed and breaking capabilities: damage of the tires and damage of the brakes. The damage of the brakes increases each time the car reduces speed, and the damage of the tires increases each time the car passes a corner faster than the recommended maximal speed.

The game uses a GASP server [**?**] for communication between the players. Originally, the logic and the rendering engine were implemented in the J2ME software, the GASP server was used only to transfer the positions between the players and a simple 2D rendering engine was used which rendered the track and the cars as sprites [**?**]. The architecture of the game is illustrated on Figure 3.6a.



(a) Original Architecture (b) Adapted Architecture

Figure 3.6: The original architecture of the car racing game (a) and the adapted architecture (b)

A conversion procedure was defined that uses the transformations presented previously (Coding, Update, Modeling and Rendering) and it consists of 5 steps:

1. Identification of relevant data for Update transformation (position of the cars, rotation of the camera in the case of the car race).

2. Defining and formalizing Scene Updates.

3. Discarding 2D graphics assets and replacement by 3D ones (Coding Transform).

4. Converting the game scene-graph into MPEG-4.

5. Discarding rendering calls and replacement by network communication (separation of the scene graph between client and server).

The MPEG-4 player presented in Section I.7 was enriched with a BIFS communication layer to ensure connection to the game server. It receives the BIFS-commands and updates the scene accordingly, and it transmits the key pressed by the player to the game server. The key-presses are detected by using the BIFS node InputSensor. When this node is activated, some JavaScript [34] code stored in the scene is executed. The code makes an HTTP request by using AJAX [2] which transfers the pressed key code to the server.

After this adaptation, the structure of the game architecture changed as illustrated on Figure 3.6b. Three main components can be observed: Client, Game server and GASP Server. The client is the MPEG-4 player, i.e. the decoding and rendering component of the distributed architecture (Figure 2.7). The Game server executes the game logic and represents the server part of the distributed architecture, i.e. the scene updates component (**U**), the Simplification component (**S**) and the part of the Coding component

(**C**) responsible for encoding and transmission. The GASP Server remains the same as in the original architecture.

The next section presents the game design and the BIFS scene-graph for each game state.

I.1.2.2 Game Design

A game design compliant to the proposed architecture has three phases: initialization, assets transmission and playing. The following paragraph defines the three phases for the car race game in two configurations, single and multi-user by analyzing at each step the transmitted data.

1. Initialization:
   - The player initiates a game by pointing to an URL referring to a remote MPEG-4 file
   - When a request is received, the server initiates a TCP session and sends the initial scene containing a simple 2D scene presenting a menu with different options: "New Game", "Connect", "Exit" as illustrated in Figure 3.7a; (the data transmitted is about 0.4 kB)
2. Assets transmission:
   - if "New Game" is selected
     a) The server sends an update command for displaying the snapshots (still images) of several cars, one at a time. The user can go forward and back trough all cars. The transmitted data is about 30 kB for 4 cars.
     b) When the user selects one snapshot, the server sends a scene update containing the 3D representation of the car (around 82 kB). The car is received by the player and rendered as an 3D object as illustrated in Figure 3.7b (the local 3D camera is also enabled).
     c) When the car is validated, the server sends still images with available circuits and the same scenario as for the car is implemented (about 30 kB for the 4 tracks, 208 kB and 1.6 MB for the 3D circuits).
   - If "Connect" is selected, the server checks for existent sessions (previously initiated by other users) and sends a list;
     a) after selecting one session (implicitly the track is selected), the server sends the screen for selecting the car;
     b) after selecting the car, the 3D object representing it is transmitted.
3. Playing the game:
   - the code of the key pressed by the user (accelerate or break) is directly transferred to the server (about 350 bps), that computes car speed and implements the rules for game logic (points, tire usage ...);
   - the server sends the new 3D position of all the cars in the race (about 650 bps), updates for the status icons and the number of points; the player processes the local scene updates as illustrated in Figure 3.8.

(a) Phase 1: Initialization

(b) Phase 2: Assets transmission - car

(c) Phase 2: Assets transmission - circuit

Figure 3.7: Snapshot from the car game (Phases 1 and 2)

(a) Phase 3: Playing the game

(b) Phase 3: Playing the game

Figure 3.8: Snapshot from the car game (Phase 3)

I.1.2.3 Description of the Scene-Graph

The game is separated in three main screens: Main Screen, Configuration Screen and Gameplay Screen. Their appearance and functionality is presented in the next sections.

I.1.2.3.1 Sending Key Actions to the Server

Common for all screens is the manner of sending the key events from the client to the server. As it was mentioned in the previous section, this is done by calling a JavaScript function that executes an AJAX request. The BIFS code can be separated in two parts: a first part for getting the key code and a second one for sending it to the server.

The key code is read by integrating an *InputSensor* node in the scene, and adding an input stream to the MPEG-4 file of type *KeySensor* attached to this node. The following code represents the integration of the InputSensor node:

```
DEF N1 Valuator {}
```

```
InputSensor {
 url [od:10]
 buffer {
  REPLACE N1.inSFInt32 BY 0
 }
}
```

The *url* field points to the stream that generates the key data. The key code is generated by the player and sent to the scene by using the stream. The buffer field receives the data and executes the REPLACE command using the received value in the first byte, which is the key code of the pressed key. Therefore the *Valuator* node will receive the key code value in its *inSFInt32* field.

The following code defines the script that sends the key to the server:

```
DEF SEND_KEY Script
{
 eventIn SFInt32 KeyCode

 url "javascript:
 function KeyCode(value)
 {
  xml = new XMLHttpRequest();
  var link = '';
  link='http://192.168.0.1:4734/';
  link=link+value;
  xml.open('GET', link, true);
  xml.send(null);
 }
 "
}
```

The *eventIn* field receives the key code and executes the *KeyCode* function using it as an input value, hence the function sends it to the server. The connection between the *Valuator* field and the script is done by defining a *ROUTE* as follows:

```
ROUTE N1.outSFInt32 TO SEND_KEY.KeyCode
```

I.1.2.3.2 Main Screen

This section represents the initial screen that is displayed when the game is loaded. The screen presents a menu with three options: New Game, Connect and Quit. The option *New Game* starts a new game session, and leads to the second screen, i.e. Configuration. The option *Connect* connects to an already existing game session and, like the first option, leads to the *Configuration* screen. The third option quits the game.

The complete scene-graph is presented in Appendix I.1. Here we will present the most important nodes and their role.

As it can be observed from the scene-graph, three similar sections exist, each for one item of the menu, consisting of six nodes:

- Transform2D - Used to position the menu item on the screen.

- Shape - General grouping node that is displayed containing appearance and geometry fields.
- Appearance - Used to reference material and textures.
- Material2D - Defines the color of the text.
- Text - Defines the string to be displayed and points to its style.
- FontStyle - Defines the size of the font and its justification.

The menu item that is selected has a bigger font size, hence the FontStyle nodes have been DEF-ed, meaning that an ID is attached to them so that they can be referenced in update commands. When the user presses the up or down key, the server changes the selected item and send a BIFS update command to change the displayed text. For example, to select the *Connect* menu item, the following update command is sent:

```
AT 2000 {
 REPLACE MENU_NEW.size BY 20
 REPLACE MENU_CONN.size BY 26
}
```

When the user selects a menu item and presses the select key, the server sends a scene replace command that loads a new scene, dependent on the selection. For example:

```
AT 3000 {
 REPLACE SCENE BY OrderedGroup {
 ....
 }
}
```

If the user selects the *Connect* command, then a list of active sessions is displayed. The list has the same structure as the main menu.

I.1.2.3.3 Configuration Screen

The configuration screen is used to select a car that will be used for racing. It consists of two screens: the first one is used to select a car from several options, and the second one is used to view the selected car in 3D. The BIFS scene-graph representing these configuration screens are presented in Appendix I.2.1 for the first one and Appendix I.2.2 for the second one.

Both screens are composed of two sections each: a menu-like user interface, and an asset display. This is implemented with Layer nodes. The menu items are implemented in the same manner as in the main menu, having options to: view previous and next car, select the current car and go back to the previous screen. The asset display section includes an image of the currently selected car. A few new nodes are woth noticing:

- Layer2D - Specifies that its children nodes should be rendered using 2D display.
- ImageTexture - Specifies a texture for the current object.
- Rectangle - Specifies that the rendered object is a rectangle with some size.

The ImageTexture node is the one used to display the image of the currently selected car.

After the user selects the *Select* menu item, and presses the select key, the second screen is loaded by using a scene replace command. In this screen the menu is changed, containing only items for confirming the selected car and to go back to the previous screen. However, the second section is different, since the Layer2D node is replaced by a Layer3D node containing the 3D model of the car. The three following nodes are introduced:

- Layer3D - Specifies that its children nodes should be rendered using 3D display.
- NavigationInfo - Specifies the type of camera used to view the 3D scene.
- Inline - Specifies that in this place in the scene-graph a new content should be included.

The type of the selected camera is *Examine*, which is a camera that rotates around a fixed point the world, usually located in the center of the object, hence it is used to examine the object from all sides. The *Inline* node includes the 3D model of the car, which is an MPEG-4 file that contains the mesh and the textures of the model.

If the user selected the *Connect* menu item in the main menu, choosing the *Select* menu item changes to the Gameplay screen. However, if the *New Game* menu item was selected, choosing the *Select* menu item changes to the track selection screen. This screen is same as for the car selection screen, the only difference being that the assets are racing tracks. At the end of the second screen it switches to the Gameplay screen.

I.1.2.3.4 Gameplay Screen

This screen is used to play the game. It is divided in two main sections: information and game, as presented in Appendix I.3.

The information section displays the number of players, the current lap, as well as information about the state of the tires and brakes. This state is presented as rectangle that is colored depending of the level of damage: green for none, yellow for medium and red for high damage. The color is changed when the server decides that there is a damage by updating the material of the appropriate rectangle, hence the materials are DEF-ed. For example, changing the color to red is done by sending the update command:

```
AT 4200 {
 REPLACE MAT_BR.emmisive BY 1 0 0
}
```

When one lap is finished, the server sends field update command to the *Text* TXT_LAPS to update the counter.

The game content, i.e. track and cars, are grouped together in a *Layer3D* node, however the cars are further wrapped in a Transform node that is DEF-ed. They are included in the scene by using the *Inline* node, however they are loaded previously when they were selected in the previous screens.

Updating the car positions is done by updating the translation field of their *Transform* nodes, hence the server generates and sends field update commands for each car when its position changes, as illustrated by the following command:

Table 3.1: Latency (transmission and decoding) for the 3D assets used in the car race game

Asset	Car_v1	Car_v2	Circuit_v1	Circuit_v2
Number of vertices	253	552	1286	7243
MPEG-4 file size (KB)	82	422	208	1600
Transmission time Wi-Fi (ms)	27	126	68	542
Transmission time UMTS (ms)	422	2178	1067	8246
Decoding time (ms)	112	219	328	2538
Total waiting time Wi-Fi (ms)	139	345	396	3080
Total waiting time UMTS (ms)	534	2397	1395	10784

```
AT 4200 {
 REPLACE TR_CAR1.translation BY 26 15 0
 REPLACE TR_CAR2.translation BY 6 35 0
}
```

I.1.2.4 Simulation

Several experiments were set up to objectively measure the user experience when playing the car race game, based on time to respond at user interaction (phases 1 and 3) and time to wait for transmission and loading of 3D assets (phase 2). Table 3.1 presents the latency when assets are transmitted and Figure 3.9 the latency when only user interaction and updates commands are transmitted. The measurements are performed for two network configurations: Wi-Fi (IEEE 802.11g, ISO/CEI 8802-11) and UMTS.

Let us note an average execution time of 80 ms and a maximum of 170 ms for Wi-Fi connection and an average of 350 ms and a maximum of 405 ms for UMTS connection for the entire loop consisting in transmission of user interaction (interface 2 in Figure 2.9), time for processing on the server, BIFS-commands transmission (interface 1 in Figure 2.9), decoding and rendering of the local scene updates.

I.1.3 Results

To evaluate the results, they were compared to the research done by Claypool M. [26] on the effect of latency on users in on-line games. The paper proposes experiments with different types of games [16, 24, 52, 59] and evaluates how the latency influences the game-play results. The quality of the game-play specific for each game is measured. For example, for first person game the hit fraction when shooting at a moving target is measured, for a driving game the lap time, etc. Then the results are normalized in a range

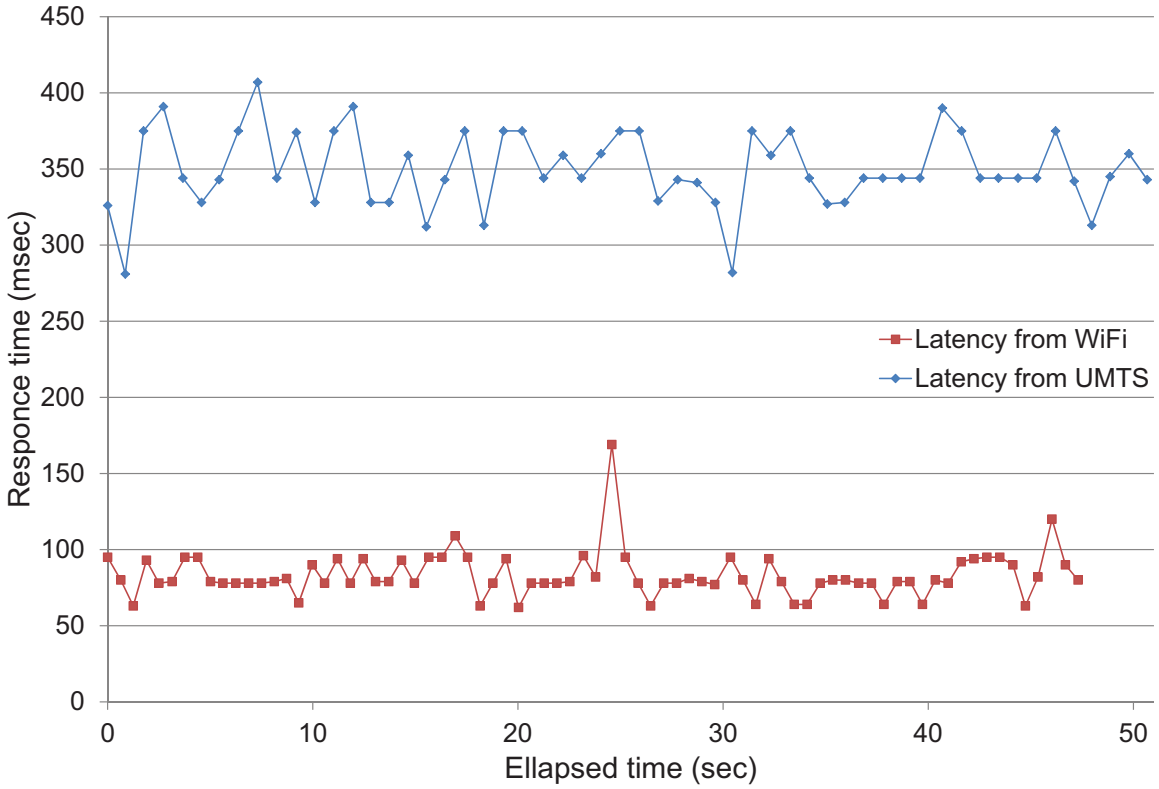


Figure 3.9: Response time (in ms) for UMTS and Wi-Fi recorded during Phase 3 "Playing the game". The horizontal axis represents the playing time

from 0 (worst) to 1 (best) and exponential curves for each game category are obtained. A copy of these curves is represented in Figure 3.10. The horizontal gray bar around 0.75 (originally proposed by Claypool M. [26]) represents the player tolerance threshold with respect to latency. The section above is the area where the latency does not affect the game play performance. The section below is the area where the game cannot be correctly played.

The latency results obtained for the race game are plotted on the same graph. Based on Figure 3.10 and on the classification of the games according to scene complexity and latency, some conclusions can be withdrawn:

- The architecture is appropriate for omnipresent games for the two network configurations.
- For third-person avatar games, the proposed architecture is appropriate when a Wi-Fi connection is used, and it is on the edge of user tolerance when a UMTS connection is used.
- For first-person avatar games, the proposed architecture is inappropriate when using UMTS connection and it is on the lower boundary of the user tolerance when using Wi-Fi connection.

It should be noted that the network conditions during the measurements were not taken into account. The goal of this experiment was to verify that it is possible to use this architecture for playing games. The research on the stability of the network conditions is out of the scope of this thesis because, as illustrated in Figure 3.10, the effect of latency

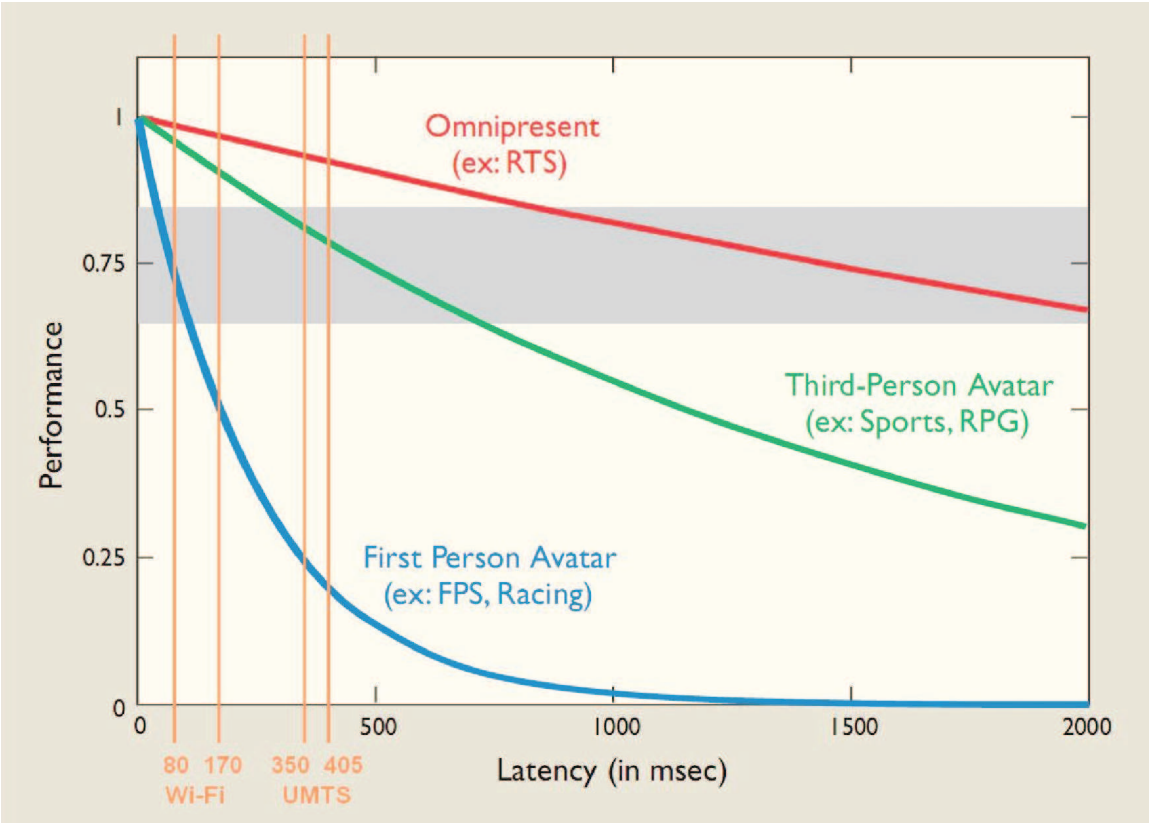


Figure 3.10: Player performance versus latency for different game categories. Original image reproduced from [26] with permission.

influences the types of game that can be played using this architecture. Increasing or decreasing the latency only means a restriction on the types of games, but not on the architecture itself.

I.2 MPEG-4 Player Architecture for Powerful Platforms

In section I.5 an optimized MPEG-4 player architecture was presented. As it can be observed, the architecture is very complex. Therefore, in order to better understand how the player works, the next section describes in more detail how different MPEG-4 streams are loaded.

I.2.1 Examples of Decoding Files

This section presents in more detail the operation of the player in different use-case scenarios that have been chosen to represent each aspect of its functionality. The following scenarios will be described:

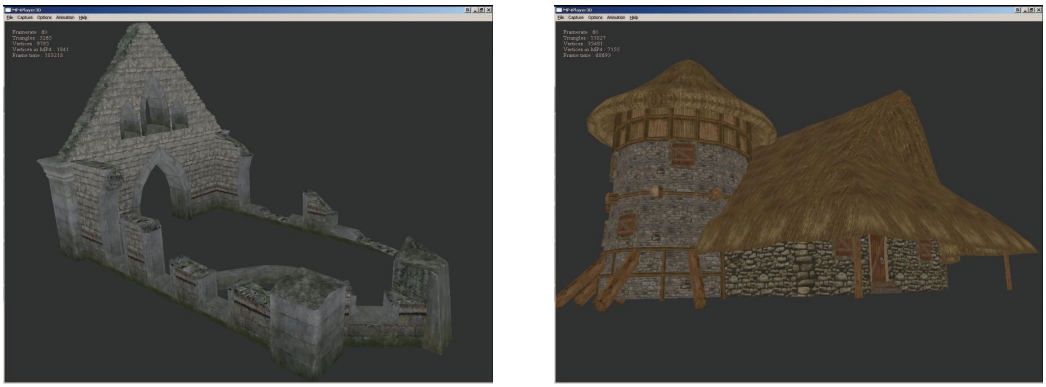
- Local Static File;
- Local Animated File;
- Local File with Streamed Animation;
- Completely Streamed File.

I.2.1.1 Local Static File

After a file open is requested, the Scene Manager calls the Stream Creator that in turn creates an DataInput instance of the type DataInputMP4. Then the Scene Manager initializes the DataInput and runs the DataInput thread that opens the file and processes the information for the streams. Then for each of the streams it calls the StreamCreator to create a Decoding Chain for that stream that includes Decoder, DataOutput and Buffers. The Stream Creator runs the Decoders and the DataOutput threads that start waiting for data.

Having a static file means that a BIFS stream is present in the file, hence the DataInput enables decoding only for the BIFS stream. After BIFS is decoded, the BIFS Data Output (DataOutputBifs) receives the Scene-Graph and creates groups(IMeshGroup) of entities (IMeshEnitites) for each mesh node in it. These are then passed to the Data Management that in turn passes them to the Renderer. If there are other resources (e.g. textures) connected with the mesh data, it connects the Data Output of the their streams to the entities and requests their decoding from the DataInput. After a texture is decoded, it will be connected to the appropriate mesh. Since the file is static, the *Timer* has no effect.

Figure 3.11 presents loaded MPEG-4 static files.



(a) Chapel (b) Cottage

Figure 3.11: Example of static MPEG-4 files

I.2.1.2 Local Animated File

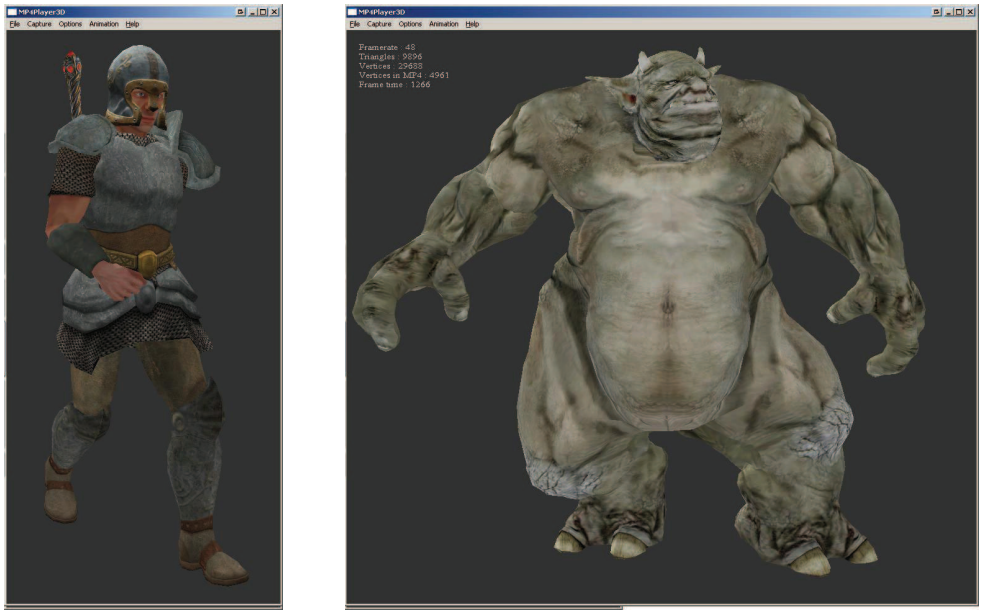
A local animated file is processed in a similar way to a static file. One difference is that BIFS may or may not be present inside the file.

If BIFS is present, the processing is done as for a static file, however some meshes may be connected to a BBA animation stream. Therefore, a *Deformer*, which receives the animation data, is created for that mesh. At each frame the *Data Output* of the BBA stream checks for data unit in the *Output Buffer* that has the current time-stamp. If one exists, it is taken out of the buffer and it is used to update the *Deformer*.

On the other hand, if BIFS is not present, it enables decoding for all streams (usually only video and/or audio). If a stream is an Image or a Video, the data output requests a rectangle mesh from the data input that is perpendicular to the camera with a specific aspect ratio. After the rectangle mesh is created, its texture is connected to the stream that requested it, hence it will display the image/video. If the stream is an audio, the

output is rendered to the sound card. As for the BBA stream, new data is checked for at each frame.

Figure 3.12 presents loaded MPEG-4 animated files with local animation.



(a) Hero (b) Troll

Figure 3.12: Example of animated MPEG-4 files with local animation

I.2.1.3 Local File with Streamed Animation

Differently from what happens in the case of the previous file, in this file the BBA animation for the mesh is not stored in the file itself, but in a remote location. The location is specified in the URL field of the SBVCAnimation node in the case of BIFS. Accessing the remote data can be done using different protocols, however the current implementation only supports RTSP (Real Time Streaming Protocol). All other data, like mesh and textures, remain in the file.

The opening of the file is same as for the previous section, however the way the BBA stream is connected to the mesh is different. When the BIFS data output detects the URL, it calls the *Stream Creator* to create the appropriate *DataInput* that connects to the remote server and receives information about the stream. Then this information is used to initialize the decoding chain for the stream. Since the BIFS data output has information for the stream type in advance, it connects the stream data output to the mesh deformer.

Figure 3.13 presents a locally loaded MPEG-4 animated file using streamed animation. The system generates cued speech movements based on the entered text by each of the users, and streams the animations to the client.

I.2.1.4 Full Streamed File

To open a scene from a stream, the user enters a URL to the server where the stream is located. RTSP is the only protocol that is supported.

Figure 3.13: Example of animated MPEG-4 files with streamed animation

At the beginning, the the Scene Manager calls the Stream Creator that in turn creates an DataInput instance of the type DataInputRTSP and runs its thread. The data input connects to the RTSP session an receives the SDP (Session Description Protocol) data which informs it about all sub-sessions, i.e. data streams, that are available. If needed, the information includes OD_IDs for each of the streams. As for a file, BIFS stream is optional.

If BIFS is present, then its sub-session is the only one that is decoded at the beginning. The other sub-sessions are decoded only at a request from the BIFS data output. Since the BIFS data is essential for the scene, it is transported in a more secure transport protocol: RTP (Real-time Transport Protocol) over TCP (Transmission Control Protocol). The textures that use a non error-resilient compression (e.g. JPEG) use the same transport protocol. For the rest of the stream for which is allowed to lose packets, RTP over UDP (User Datagram Protocol) is used.

Internally the decoding of the data is done in the same manner as for a static file.

Figure 3.14 presents a completely streamed MPEG-4 animated file with streamed animation. When the website is opened, the MPEG-4 player requests the file from the server, which is then streamed. Furthermore, the animation stream is loaded from a system that generates cued speech movements based on the entered text by the user.

I.2.2 On-line Animation System

In order to validate the player architecture for a powerful platform, an on-line system for CS (Cued Speech) was designed, able to synthesize in real time face and hand animation for CS, based on the text or speech input by the user. The novelty of our approach consists in the system architecture based on a dedicated server able to perform costly operations and to deliver the results as a compressed animation stream. On the user side, only a 3D graphics player is required, being possible to implement it on light terminals. Such an approach has the advantage of processing the speech very close to the capture place, avoiding voice quality loss due to transmission errors and bandwidth. The proposed architecture is illustrated in Figure 3.15. On the server side, the two entries, voice and text, are converted in animation parameters. The latter are encoded as an MPEG-4

Figure 3.14: Example of completely streamed MPEG-4 file

animation stream and broadcast to the network. On the client side, an MPEG-4 player receives the animation stream and updates, in a continuous manner, the scene graph defining the avatar's face and hand.

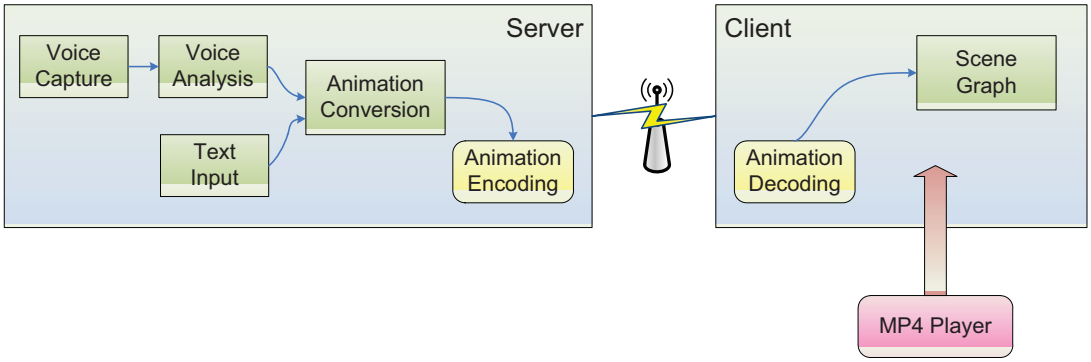


Figure 3.15: Proposed Architecture for the Online Cued Speech system

The system is composed of three main components: Production, Transmission and Visualization.

I.2.2.1 Production

In the production component there are two modules: off-line preparation of the avatar and on-line animation generation from speech and text. The first is based on a protocol establishing the way to build the configurations for the face and for the hand. The output is an MPEG-4 file defining the avatar. In our model, the face is defined by 11 target shapes (Figure 3.16); for the hand there are 9 configurations and 6 positions.

The on-line animation production exploits the previously obtained MPEG-4 file and converts each phoneme into MPEG-4 animation parameters. These parameters are then compressed by the BBA encoder.

Figure 3.16: Different target shapes defining the morph space

I.2.2.2 Transmission

The output of the BBA encoder provides animation data encapsulated in standard AUs (Access Units). Each AU contains time information that can be used for obtaining transport packets. Two transport protocols are currently implemented: UDP and RTSP.

I.2.2.3 Visualization

On the client size, the MPEG-4 3D Graphics player is able to load a local or remote file or stream, decode the geometry, the texture and the animation, and render the 3D graphics scene.

I.2.2.4 Implementation Examples

Based on the components presented above, two prototypes were developed for learning and practicing CS: a web service where the user inputs the text and visualizes the animation, and a chat service, allowing a CS communication between two users. In both prototypes, a server-client architecture is used, where the server has the role of computing and encoding the MPEG-4 animation parameters, and the client has only the role of decoding and visualizing the avatar. Together with the animation, a sound track is computed (synthesized from text), compressed (in MPEG-4 AAC), transmitted and played by the same MPEG-4 player. Since similar approaches are used in both prototypes, only the components of the Chat service is presented in details. The architecture of the Chat service for two clients is illustrated in Figure 3.17. The two main components of the system are the AS (Application Server) and the CC (Chat Client). The AS manage the communication between the clients and the conversion from text to speech and animation. The CC gets the input from the user, sends it to the AS as ASCII text, decodes and displays the animation and the audio received from the server. The AS is composed of the following units: Chat server, TTS (Text To Speech) and CS engine, CS to BBA converter, WAV to AAC converter and RTSP streaming server. The Chat server is responsible for managing the communication session (login, authentication) and the text exchanges between the clients. The text messages received from the clients are first sent to the TTS and CS engine, and when the synthesized audio and animation are ready for streaming, it sends the text to the other clients. The TTS and LPC engine is used to convert the text to synthesized audio and CS commands for animation.

The CS data and the audio are converted into more usable streams. The CS is converted into a BBA stream, which can be used directly by the MPEG-4 player to display

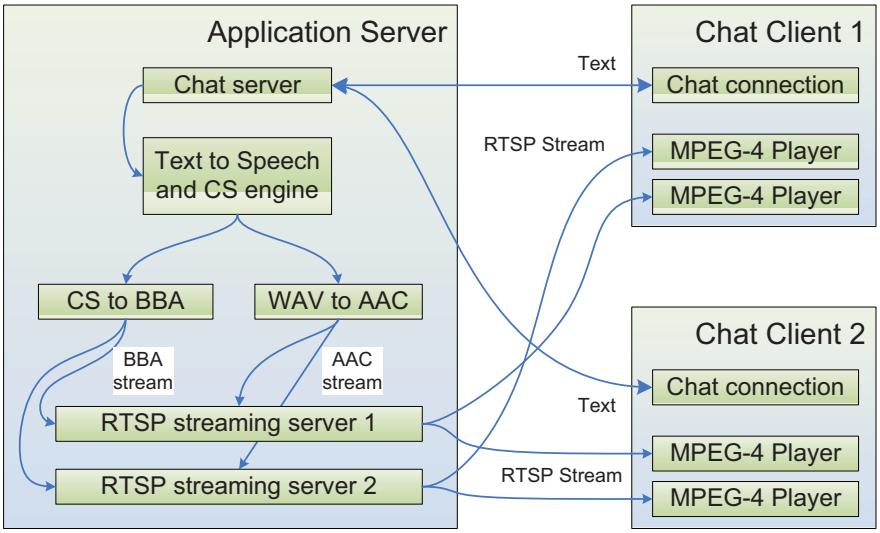


Figure 3.17: The architecture of the Chat service

the animation. The audio is compressed into an AAC stream. Then both bit-streams are sent to the RTSP servers, which in turn send them to the appropriate Client. The use of the RTSP server is needed to achieve synchronization between the animation and the sound. There is a separate RTSP server for each connected user. Therefore the chat client initializes a new RTSP server for each new connected user. The Live555 library is used to implement the RTSP server and it was extended to be able to read and synchronize the BBA and AAC streams. The CC is composed of the following units: Chat connection (Figure 3.13) and MPEG-4 player. The process of connecting to the Chat server is performed in the following manner: after the user authenticates, he joins the chat-room and he can send and receive text. Additionally the user receives from the server the parameters for initializing the RTSP connection. The client initializes the MPEG-4 player that loads a local MPEG-4 file containing the avatar. After connection to the streaming server, it is ready to receive animation and audio data. When the user enters text, the text is sent to the server, and all the connected users receive the entered text, the animation and the audio stream.

I.3 MPEG Extensible Middleware (MXM)

Section I.6 proposed an API for accessing MPEG-4 content by using only a few functions and without any knowledge of the MPEG-4 standard. In order to validate the API, a MPEG-4 file viewer was implemented using the Ogre 3D [1] rendering engine, which has its own proprietary file format for storing 3D mesh data and materials, but which it exposes complete structures for loading data from other formats.

Loading an MPEG-4 file requires a few steps:

1. Creating an engine to access the methods:

```
MediaFrameworkEngine::Access::Engine eng;
```

[1] Ogre 3D rendering engine: http://www.ogre3d.org/

2. Loading an MPEG-4 file:

```
eng.LoadMP4(fileName, MediaFrameworkEngine::Access::Engine::OUT_VB);
```

3. Parsing the appearance:

```
eng.GetGraphics3DEngine()->GetAppearance()->GetAppBuffer(true,
  buff, size);
ParseAppearance(eng, prefix, buff, size);
eng.GetGraphics3DEngine()->GetAppearance()->FreeBuffer(buff);
```

While parsing the appearance buffer, the Ogre3D specific appearance structures are created. After all appearance structures have been parsed, the needed textures are loaded. This is done by accessing the Image Engine:

```
MediaFrameworkEngine::Access::Image::ImageAccess::ImageDescriptor d;
  eng.GetImageEngine()->GetDecodedImage(true, texIds[i], texbuff,
  texsize, d);
```

The parameter d returns the size of the image and its pixel format

4. Parsing the vertex and index buffers:

```
eng.GetGraphics3DEngine()->GetGeometry()->GetVBandIB(true,
  buff, size);
ParseVB(List, eng, prefix, buff, size);
eng.GetGraphics3DEngine()->GetGeometry()->FreeBuffer(buff);
```

While parsing the vertex and index buffers, the Ogre3D specific mesh structures are created. Furthermore, each mesh is connected to the correct appearance structure by using the id provided in each index buffer.

As it can be observed from the previous example, the integration of the MXM API in a third party 3D engine becomes an easy exercise. Figure 3.18 presents a screen shot from a running Ogre3D base MPEG-4 player that has many loaded files.

I.4 Conclusion

This chapter presented the experiments and the validation for the proposed distributed architecture as well as the architectures for the MPEG-4 players for powerful platform and mobile devices.

The first part of this chapter presented experiments and results with relation to the alternative architecture for games on mobile phones aiming to cope with the problems of costly deployment of games due to the heterogeneity of the mobile terminals. By exploiting and extending some concepts of "thin clients" introduced in the early nineties,

Figure 3.18: Ogre3D based player with many MPEG-4 files loaded

the solution was based on 3D graphics capabilities of MPEG-4. An implementation of an MPEG-4 3D graphics player for mobile phones was presented and its capabilities in terms of decoding time and rendering performance were evaluated. It was observed that in order to achieve reasonable decoding times, the maximum size of one static object should be 15000 vertices and for an animated one it should be 1500 vertices. However, in order to achieve reasonable frame-rates of around 25 fps the total number of rendered vertices should be less than 20000 for static objects and less than 6000 for animated ones. For scenes that combine both types of objects, the number of vertices should be calculated proportionally. Furthermore, a pre-processing technique was presented that reduced the size of the data and its decoding time for around 15%. Additionally, it was presented that applying a simplification algorithm on the models can decrease their size, decoding and rendering complexity while preserving similar visual appearance.

Demonstrating the pertinence of using MPEG-4 as a standard solution of representing 3D assets on mobile phones, the main idea of the proposed architecture is to maintain the rendering on the user terminal and to move the game logic to a dedicated server. The approach presents several advantages, such as the ability to play the same game on different platforms without adapting and recompiling the game logic, and to use the same rendering engine to play different games. The usage of MPEG-4 scene updates for the communication layer between the game logic and the rendering engine allows the independent development of the two, with the advantage of breaking the constraints on the game performances (by using the server) and of addressing a very fragmented and heterogeneous park of mobile phones. The measurements performed on different networks and end-user terminals are satisfactory with respect to the quality of player gaming experience. In particular, the latency between user interaction on the keyboard and the rendered image is small enough (around 80 ms for Wi-Fi and 350 ms for UMTS) to ensure that a large category of games can be played on the proposed architecture.

In the next part of the chapter, the architecture of the MPEG-4 player for powerful platform was evaluated. First it was presented how different type of content is loaded, including local static file, local animated file, local file with streamed animation and completely streamed file. Then the architecture was further validated by implementing

an on-line animations system. The system used a server for generating cued speech animation and speech from inputed text. Two use case scenarios were presented: a web application for learning and practicing cued speech and a chat program. Both scenarios validated that the player architecture is capable of supporting complex MPEG-4 content. For the powerful platform, the next section validated the MXM API, which is composed of many smaller APIs, i.e. engines, focused each on one part of the MPEG-4 standard. It was confirmed that integrating this API into a third party 3D rendering engine is an easy task by creating an application that can load a display 3D graphics content using the Ogre3D 3D graphics rendering engine.

Conclusion and Perspectives

The main topic of this thesis is to propose an alternative client-server architecture for creating mobile games where the connectedness of the mobile devices is used. The newer 3D games demand a lot of processing power, both for general and graphics processing. However mobile devices are not capable to effectively execute everything, hence the application has to find a balance between what is on the client vs. the server. A distributed solution was proposed, where the game logic is executed on the server and the rendering is executed on the client. For this solution to be effective, three main requirements were defined:

1. Minimize the network traffic and reduce data rate fluctuations,
2. Reduce the required computational power on the terminal, and
3. Preserve the user experience compared with local execution.

First, the development of distributed computer architectures was analyzed. Furthermore, the advance in modern remote computing was presented, focusing on Internet applications. Then the current state of the research connected to remote computing was presented. Different architectures were analyzed considering the requirements. However it was observed that none of them could satisfy all requirements, thus the need for a new solution was detected.

It was observed that a possible solution is to standardize a component that is common for all games, namely the scene-graph. By using a standard scene-graph format, it becomes possible to standardize the client application, as well as the communication protocol. Therefore different standards were analyzed and compared. The analysis focused on their capabilities to support 3D graphics assets, streaming and compression and user interactions. The MPEG-4 format was observed as the only one that integrates all needed features, hence it was chosen to be used in the architecture.

In order to analyze the architectures presented in the state of the art review, a framework of functions was defined, which represents each processing stage. This mathematical representation enabled having an overview of the architecture in terms of limitations and use-case scenarios. It was observed that none of them is appropriate for use in the targeted applications of this thesis. Therefore, a new framework was proposed, all its components were presented and solutions were proposed for each of them. The architecture uses the MPEG-4 standard as a scene-graph format for the game, as well as a communication protocol between the server and the client. The client to server data (i.e. user commands) is transferred using AJAX requests.

The next section investigated a design of an architecture intended for an MPEG-4 player on a powerful platform. It presented an optimization of the MPEG-4 System Decoder Model that allows loading of MPEG-4 files having different scene-graph formalisms. Furthermore, threading is used to execute the decoding tasks in parallel, as well as allowing easier management of input from different streams at the same time. The architecture was validated by implementing an MPEG-4 player capable of supporting files with different media (3D graphics, animation, video and sound) as well as distributed applications. A web application was presented that uses a server for converting text to audio and cued speech animations.

However, it was observed that implementing this architecture is complex, which can be overlooked if one needs an MPEG-4 player. Because some applications may use MPEG-4 only for storing assets and therefore not require all features of the standard, a simplified access mechanism is necessary. Hence, the MPEG Extensible Middleware was designed,

which is a simple API for accessing MPEG-4 files. We propose a 3D Graphics Engine API and an example of its usage was also presented, using the Ogre 3D engine. This proves that the API was simple enough to be used in third party architectures.

Because the mobile devices have less processing capabilities than the PCs, the player architecture had to be changed to reflect these restrictions. Furthermore, other restrictions had to be implemented: limiting to only one scene-graph format (i.e. BIFS), limiting the number of used nodes to the minimum needed and limiting the supported codecs. This allowed having a more optimized player that can be run on a mobile device.

Two types of experiments were performed: decoding and rendering for both static and animated content that showed satisfying results. However, games require, apart from rendering, also other processing. To be able to support both, the maximum number of rendered vertices has to be reduced, thus the quality of the rendered image will also be reduced. Therefore, a client-server architecture was proposed that can solve this problem by executing the game logic on the server, and only the rendering on the client.

Because the features of the MPEG-4 standard include not only representation of a scene, but also scene updating, the standard was used to represent the scene-graph of the game, as well as to define the communication protocol between the two components. Since a standard protocol is used, both components can be developed independently. On one hand, a server (i.e. game) can be developed without the knowledge of the client, meaning that different clients, for different mobile devices can be used for the same server. On the other hand, a client can be developed without any knowledge of a particular game, and furthermore optimized for a specific mobile device, but remaining capable of playing the same game.

To demonstrate the architecture, an existing game was adapted, which consist of simple car racing by controlling only the speed of the car. A few experiments were performed, measuring the bandwidth and the round-trip time (time between the user pressing a key, and the player receiving updated data from the server) on UMTS and Wi-Fi networks. The results were compared to the research done by Claypool M. [26] on the effect of latency on users in on-line games. Based on the results and on the classification of the games according to scene complexity and latency, some conclusions were withdrawn:

- The architecture is appropriate for omnipresent games for the two network configurations,
- For third-person avatar games, the proposed architecture is appropriate when a Wi-Fi connection is used, and it is on the edge of user tolerance when a UMTS connection is used,
- For first-person avatar games, the proposed architecture is inappropriate when using UMTS connection and it is on the lower boundary of the user tolerance when using Wi-Fi connection.

To summarize, the architecture can be used for a wide range of games, however they should be designed to use it from the beginning.

Perspectives

The perspectives for the work in this thesis can be sought from the broader viewpoint of distributed systems for 3D graphics. On one side are architectures that render everything on the client, while on the other side are architectures that render everything on the server and transfer video to the client. As it was presented in Chapter I, there are already techniques that fit in-between these extreme cases, and depending on the requirements of an application, an appropriate architecture can be selected. However, the state of the art does not cover all possible combinations.

This thesis handled one of these gaps, where the performance on the terminal allowed rendering, but not other operations. Future work may focus on the video streaming architectures where, for example, using the input of the rendering process may improve the rendering performance both in processing time and quality. This can be done by utilizing the information for the main focus object of the scene to choose the encoding parameters for having better quality, while the background is encoded with worse quality. Furthermore, an optimization of the video encoding process can be achieved by using the Z buffer output of the rendering process to optimize the detection of the macro blocks for the video encoding.

Having most of the use-cases handled by different architectures can be used to design an architecture that can switch dynamically from one distributed system to another depending on the environment. Some research has already been carried in this direction, which was presented in Chapter I, however these systems use only two or three architectures, and are limited to only a small portion of the whole range of possible architectures. A better system could be able to switch between wider range of distributed architectures.

Bibliography

[1] Buisson p., kozon m., raissouni a., tep s., wang d., xu l., jeu multi-joueur sur telephone mobile, (in french), rapport de projet ingenieur, rapport final projet s4 2007 (TELECOM bretagne), available online at http://proget.int-evry.fr/projects/JEMTU/ConceptReaJeu.html. [cited at p. xxv]

[2] Garrett JJ, 2005, ajax: A new approach to web applications, http://www.adaptivepath.com/ideas/essays/archives/000385.php, february 2005. [cited at p. 81]

[3] Pellerin r., delpiano f., duclos f., Gressier-Soudan e. et simatic m., GASP: an open source gaming service middleware dedicated to multiplayer games for J2ME based mobile phones, proceedings of international conference on computer games, angouleme, france, 28-30 novembre 2005. [cited at p. xxiv]

[4] ISO/IEC 14496-11:2005 - "information technology coding of audio-visual objects part 11: Scene description and application engine", 2004. [cited at p. 23, 31]

[5] ISO/IEC 14772-1:1997 and ISO/IEC 14772-2:2004 virtual reality modeling language (VRML), 2004. [cited at p. 20]

[6] ISO/IEC 19775:2004 Extensible 3d (X3D), 2004. [cited at p. 21]

[7] ISO/IEC 14496-11:2005 - "information technology coding of audio-visual objects part 1: Systems", 2005. [cited at p. 25]

[8] ISO/IEC 14496-11:2005 - "information technology coding of audio-visual objects part 16: Animation framework extension (afx)", 2005. [cited at p. 26, 29, 33]

[9] ECMAScript Language Specification, ECMA-262, ISO/IEC 16262, ecma international, 5th edition, 2009. [cited at p. 22]

[10] Ghassan Al-Regib and Yucel Altunbasak. 3TP: 3-D models transport protocol. In *Proceedings of the ninth international conference on 3D Web technology*, pages 155–162, Monterey, California, 2004. ACM. [cited at p. 13]

[11] Ghassan Al-Regib, Yucel Altunbasak, and Jarek Rossignac. Error-resilient transmission of 3D models. *ACM Trans. Graph.*, 24(2):182–208, 2005. [cited at p. 13]

[12] Matt Aranha, Piotr Dubla, Kurt Debattista, Thomas Bashford-Rogers, and Alan Chalmers. A physically-based client-server rendering solution for mobile devices. In *Proceedings of the 6th international conference on Mobile and ubiquitous multimedia*, pages 149–154, Oulu, Finland, 2007. ACM. [cited at p. 9]

[13] Remi Arnaud and Mark C. Barnes. *Collada: Sailing the Gulf of 3d Digital Content Creation*. AK Peters Ltd, 2006. [cited at p. 22]

[14] C. Bajaj, S. Cutchin, V. Pascucci, and G. Zhuang. Error resilient streaming of compressed VRML. 1998. [cited at p. 13]

[15] Ricardo A. Baratto, Leonard N. Kim, and Jason Nieh. THINC: a virtual display architecture for thin-client computing. *SIGOPS Oper. Syst. Rev.*, 39(5):277–290, 2005. [cited at p. 10]

[16] Tom Beigbeder, Rory Coughlan, Corey Lusher, John Plunkett, Emmanuel Agu, and Mark Claypool. The effects of loss and latency on user performance in unreal tournament 2003. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 144–151, Portland, Oregon, USA, 2004. ACM. [cited at p. 87]

[17] Azzedine Boukerche, Raed Jarrar, and Richard Werner Pazzi. An efficient protocol for remote virtual environment exploration on wireless mobile devices. In *Proceedings of the 4th ACM workshop on Wireless multimedia networking and performance modeling*, pages 45–52, Vancouver, British Columbia, Canada, 2008. ACM. [cited at p. 9]

[18] Ian Buck, Greg Humphreys, and Pat Hanrahan. Tracking graphics state for networked rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 87–95, Interlaken, Switzerland, 2000. ACM. [cited at p. 7]

[19] Dick C.A. Bulterman and Lloyd W. Rutledge. *SMIL 3.0: Flexible Multimedia for Web, Mobile Devices and Daisy Talking Books*. Springer Publishing Company, Incorporated, 2008. [cited at p. 21]

[20] T. Capin, K. Pulli, and T. Akenine-Moller. The state of the art in mobile graphics research. *Computer Graphics and Applications, IEEE*, 28(4):74 –84, 2008. [cited at p. xv, 17]

[21] Romain Cavagna, Christian Bouville, Patrick Gioia, and Jérôme Royan. A mpeg-4 afx compliant platform for 3d contents distribution in peer-to-peer. In *ICIP*, pages 2688–2691, 2008. [cited at p. 34]

[22] Eun-Young Chang, Namho Hur, and E.S. Jang. 3d model compression in mpeg. In *Image Processing, 2008. ICIP 2008. 15th IEEE International Conference on*, pages 2692 –2695, 12-15 2008. [cited at p. 33]

[23] Liang Cheng, Anusheel Bhushan, Renato Pajarola, and Magda El Zarki. Real-time 3D graphics streaming using MPEG-4. *In Proc. IEEE/ACM Wksp. on Broadband Wireless Services and Appl*, 2004. [cited at p. 8]

[24] Mark Claypool. The effect of latency on user performance in real-time strategy games. *Comput. Netw.*, 49(1):52–70, 2005. [cited at p. 87]

[25] Mark Claypool. Motion and scene complexity for streaming video games. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, pages 34–41, Orlando, Florida, 2009. ACM. [cited at p. 51]

[26] Mark Claypool and Kajal Claypool. Latency and player actions in online games. *Commun. ACM*, 49(11):40–45, 2006. [cited at p. xxix, 51, 87, 88, 89, 120]

[27] Volker Coors. Resource-adaptive interactive 3D maps. In *Proceedings of the 2nd international symposium on Smart graphics*, pages 140–144, Hawthorne, New York, 2002. ACM. [cited at p. 14, 16]

[28] J. Diepstraten, M. Gorke, and T. Ertl. Remote line rendering for mobile devices. In *Computer Graphics International, 2004. Proceedings*, pages 454–461, 2004. [cited at p. 11]

[29] Jean-Claude Dufourd, Olivier Avaro, and Cyril Concolato. An mpeg standard for rich media services. *IEEE MultiMedia*, 12(4):60–68, 2005. [cited at p. 23]

[30] Leigh Edwards and Richard Barker. *Developing Series 60 Applications: A Guide for Symbian OS C++ Developers*. Pearson Higher Education, 2004. [cited at p. xxiii, 69]

[31] P. Eisert and P. Fechteler. Low delay streaming of computer graphics. In *Image Processing, 2008. ICIP 2008. 15th IEEE International Conference on*, pages 2704–2707, 2008. [cited at p. 51]

[32] Klaus Engel, Ove Sommer, and Thomas Ertl. A framework for interactive hardware accelerated remote 3D-Visualization. *IN PROC. TCVG SYMP. ON VIS. (VISSYM*, pages 167—177, 2000. [cited at p. 9]

[33] Jean Le Feuvre, Cyril Concolato, and Jean-Claude Moissinac. GPAC: open source multimedia framework. In *Proceedings of the 15th international conference on Multimedia*, pages 1009–1012, Augsburg, Germany, 2007. ACM. [cited at p. xxiii, 69]

[34] David Flanagan and Flanagan David. *JavaScript: The Definitive Guide*. O'Reilly Media, 5 edition, August 2006. [cited at p. 81]

[35] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 209–216. ACM Press/Addison-Wesley Publishing Co., 1997. [cited at p. xxiv, 78]

[36] Ian J. Grimstead, Nick J. Avis, and David W. Walker. Visualization across the pond: how a wireless PDA can collaborate with million-polygon datasets via 9,000km of cable. In *Proceedings of the tenth international conference on 3D Web technology*, pages 47–56, Bangor, United Kingdom, 2005. ACM. [cited at p. 16]

[37] C. Herpel and A. Eleftheriadis. Mpeg-4 systems: Elementary stream management. *Signal Processing: Image Communication*, 15(4-5):299 – 320, 2000. [cited at p. 29]

[38] Mojtaba Hosseini and Nicolas D. Georganas. MPEG-4 BIFS streaming of large virtual environments and their animation on the web. In *Proceedings of the seventh international conference on 3D Web technology*, pages 19–25, Tempe, Arizona, USA, 2002. ACM. [cited at p. 49]

[39] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordan Stoll, Matthew Everett, and Pat Hanrahan. WireGL: a scalable graphics system for clusters. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 129–140. ACM, 2001. [cited at p. 7]

[40] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *ACM SIGGRAPH ASIA 2008 courses*, pages 1–10, Singapore, 2008. ACM. [cited at p. 8]

[41] Tom Jehaes, Peter Quax, and Wim Lamotte. Adapting a large scale networked virtual environment for display on a PDA. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 217–220, Valencia, Spain, 2005. ACM. [cited at p. 15, 16]

[42] B. Jovanova, M. Preda, and F. Preteux. MPEG-4 part 25: A generic model for 3D graphics compression. In *3DTV Conference: The True Vision - Capture, Transmission and Display of 3D Video, 2008*, pages 101–104, 2008. [cited at p. 52]

[43] A. Jurgelionis, P. Fechteler, P. Eisert, F. Bellotti, H. David, J. P. Laulajainen, R. Carmichael, V. Poulopoulos, A. Laikari, P. Perala, A. De Gloria, and C. Bouras. Platform for distributed 3D gaming. *Int. J. Comput. Games Technol.*, 2009:1–15, 2009. [cited at p. xv, 6]

[44] Khronos. Collada 1.5 specification, 2008. [cited at p. 22]

[45] Michelle Kim, Steve Wood, and Lai-Tee Cheok. Extensible mpeg-4 textual format (xmt). In *MULTIMEDIA '00: Proceedings of the 2000 ACM workshops on Multimedia*, pages 71–74, New York, NY, USA, 2000. ACM. [cited at p. 23]

[46] David Koller, Michael Turitzin, Marc Levoy, Marco Tarini, Giuseppe Croccia, Paolo Cignoni, and Roberto Scopigno. Protected interactive 3D graphics via remote rendering. *ACM Trans. Graph.*, 23(3):695–703, 2004. [cited at p. 15]

[47] Martin Kurze and Roman Englert. Network centric photorealistic mixed reality on mobile devices. In *Proceedings of the 3rd international conference on Mobile technology, applications \& systems*, page 26, Bangkok, Thailand, 2006. ACM. [cited at p. 9, 10]

[48] Fabrizio Lamberti, Claudio Zunino, Andrea Sanna, Antonino Fiume, and Marco Maniezzo. An accelerated remote graphics architecture for PDAS. In *Proceedings of the eighth international conference on 3D Web technology*, pages 55–ff, Saint Malo, France, 2003. ACM. [cited at p. 8]

[49] Khaled Mamou, Titus Zaharia, and Françoise Prêteux. Tfan: A low complexity 3d mesh compression algorithm. *Comput. Animat. Virtual Worlds*, 20(2‐3):343–354, 2009. [cited at p. 33]

[50] Francisco Moran, Marius Preda, Gauthier Lafruit, Paulo Villegas, and Robert-Paul Berretty. 3D game content distributed adaptation in heterogeneous environments. *EURASIP J. Adv. Signal Process*, 2007(2):31–31, 2007. [cited at p. 51]

[51] I. Nave, H. David, A. Shani, Y. Tzruya, A. Laikari, P. Eisert, and P. Fechteler. Games@large graphics streaming architecture. In *Consumer Electronics, 2008. ISCE 2008. IEEE International Symposium on*, pages 1–4, 2008. [cited at p. 7, 8]

[52] James Nichols and Mark Claypool. The effects of latency on online madden NFL football. In *Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*, pages 146–151, Cork, Ireland, 2004. ACM. [cited at p. 87]

[53] Antti Nurminen. m-LOMA - a mobile 3D city map. In *Proceedings of the eleventh international conference on 3D web technology*, pages 7–18, Columbia, Maryland, 2006. ACM. [cited at p. 14, 15, 16]

[54] Antti Nurminen. Mobile, hardware-accelerated urban 3D maps in 3G networks. In *Proceedings of the twelfth international conference on 3D web technology*, pages 7–16, Perugia, Italy, 2007. ACM. [cited at p. 14, 15]

[55] Marius Preda, Blagica Jovanova, Ivica Arsov, and Françoise Prêteux. Optimized mpeg-4 animation encoder for motion capture data. In *Web3D '07: Proceedings of the twelfth international conference on 3D web technology*, pages 181–190, New York, NY, USA, 2007. ACM. [cited at p. 34, 72]

[56] Marius Preda, Paulo Villegas, Franciso Moran, Gauthier Lafruit, and Robert-Paul Berretty. A model for adapting 3D graphics based on scalable coding, real-time simplification and remote rendering. *Vis. Comput.*, 24(10):881–888, 2008. [cited at p. xviii, xxiv, 39, 78]

[57] Jean-Charles Quillet, Gwenola Thomas, Xavier Granier, Pascal Guitton, and Jean-Eudes Marvie. Using expressive rendering for remote visualization of large city models. In *Proceedings of the eleventh international conference on 3D web technology*, pages 27–35, Columbia, Maryland, 2006. ACM. [cited at p. 12]

[58] Robert W. Scheifler and Jim Gettys. The x window system. *ACM Trans. Graph.*, 5(2):79–109, 1986. [cited at p. 10]

[59] Nathan Sheldon, Eric Girard, Seth Borg, Mark Claypool, and Emmanuel Agu. The effect of latency on user performance in warcraft III. In *Proceedings of the 2nd workshop on Network and system support for games*, pages 3–14, Redwood City, California, 2003. ACM. [cited at p. 87]

[60] Julien Signs, Yuval Fisher, and Alexandros Eleftheriadis. Mpeg-4's binary format for scene description. *Signal Processing: Image Communication*, 15(4-5):321 – 345, 2000. [cited at p. 31]

[61] Simon Stegmaier, Marcelo Magallon, and Thomas Ertl. A generic solution for hardware-accelerated remote visualization. In *Proceedings of the symposium on Data Visualisation 2002*, pages 87–ff, Barcelona, Spain, 2002. Eurographics Association. [cited at p. 10]

[62] S. M. Tran, M. Preda, F. J. Preteux, and K. Fazekas. Exploring MPEG-4 BIFS features for creating multimedia games. In *Proceedings of the 2003 International Conference on Multimedia and Expo - Volume 2*, pages 429–432. IEEE Computer Society, 2003. [cited at p. 49]

[63] N. Trevett. Khronos and OpenGL ES, proceedings of siggraph 04, tokyo, japan, 2004 http://www.khronos.org/opengles/1_X/. [cited at p. xxiii, 70]

[64] W3C. Scalable vector graphics (SVG). http://www.w3.org/Graphics/SVG/. [cited at p. 21]

[65] W3C. Mobile SVG profiles: SVG Tiny and SVG Basic, 2003. [cited at p. 22]

[66] D. De Winter, P. Simoens, L. Deboosere, F. De Turck, J. Moreau, B. Dhoedt, and P. Demeester. A hybrid thin-client protocol for multimedia streaming and interactive gaming applications. In *Proceedings of the 2006 international workshop on Network and operating systems support for digital audio and video*, pages 1–6, Newport, Rhode Island, 2006. ACM. [cited at p. 14]

[67] Zhidong Yan, S. Kumar, and C.-C.J. Kuo. Mesh segmentation schemes for error resilient coding of 3-D graphic models. *Circuits and Systems for Video Technology, IEEE Transactions on*, 15(1):138–144, 2005. [cited at p. 13]

Appendices

Appendix A

Related Publications

Book chapter:

1. I. Arsov, M. Preda, and F. Preteux, "A Server-Assisted Approach for Mobile-Phone Games", Mobile Multimedia Processing, 2010, pp. 170-187.

Journal articles:

1. B. Jovanova, I. Arsov, M. Preda, F. Preteux, On-line animation system for practicing Cued Speech, International Journal of Image and Graphics (IJIG), vol. 10, Issue: 4, Oct. 2010, pp. 497-512.

2. M. Preda, I. Arsov, and F. Moran, COLLADA + MPEG-4 OR X3D + MPEG-4, Vehicular Technology Magazine, IEEE, vol. 5, Mar. 2010, pp. 39-47.

Conference articles:

1. A. M. Khan, I. Arsov, M. Preda, S. Chabridon, A. Beugnard, Adaptable client-server architecture for mobile multi-player games, DIstributed SImulation & Online gaming (DISIO), Torremolinos, Spain, 2010

2. I. Arsov, B. Jovanova, M. Preda, and F. Preteux, When MPEG-4 and COLLADA meet for a complete solution of distributing and rendering 3D graphics assets, Consumer Electronics (ICCE), 2010 Digest of Technical Papers International Conference on, 2010, pp. 431 -432.

3. I. Arsov, B. Jovanova, M. Preda, and F. Preteux, On-Line Animation System for Learning and Practice Cued Speech, ICT Innovations 2009, pp. 315-325.

4. I. Arsov, M. Preda, and F.J. Preteux, MPEG-4 3D graphics for mobile phones, WMMP, 2008.

5. M. Preda, B. Jovanova, I. Arsov, and F. Preteux, Optimized MPEG-4 animation encoder for motion capture data, Proceedings of the twelfth international conference on 3D web technology, Perugia, Italy: ACM, 2007, pp. 181-190.

Standardization reports:

1. I. Arsov, M. Preda, Using MPEG-4 for mobile mixed reality applications Standardization Report ISO/IEC JTC1/SC29/WG11, MPEG2011/M19289, Daegu, Korea, January 2011

2. I. Arsov, L.V. Ngo, M. Preda, FootPrint API for MXM, Standardization Report ISO/IEC JTC1/SC29/WG11, MPEG2009/M17284, Kyoto, Japan, January 2010

3. I. Arsov, M. Preda, F. Preteux, Integrated MXM API for 3D Graphics, Standardization Report ISO/IEC JTC1/SC29/WG11, MPEG2009/M16427, Maui, USA, April 2009

4. I. Arsov, M. Preda, F. Preteux, MXM API for 3D Graphics content creation, Standardization Report ISO/IEC JTC1/SC29/WG11, MPEG2009/M16151, Lausanne, Switzerland, February 2009

5. I. Arsov, M. Preda, F. Preteux, MPEG-4 3D Graphics Player for N93 and N95, Standardization Report ISO/IEC JTC1/SC29/WG11, MPEG2008/M15087, Antalya, Turkey, January 2008.

6. I. Arsov, M. Preda, F. Preteux, Support for multiple texture coordinates and additional attributes per vertex in 3DMC, Standardization Report ISO/IEC JTC1/SC29/WG11, MPEG2007/M14905, Shenzhen, China, October 2007.

7. M. Preda, I. Arsov, B. Jovanova, F. Preteux, Compression performances of MPEG-4 3D Graphics for large databases, Standardization Report ISO/IEC JTC1/SC29/WG11, MPEG2007/M14710, Lausanne, Switzerland, July 2007.

8. M. Preda, T. Laquet, I. Arsov, C. Pelvet, O. Marre, F. Preteux, MPEG-4 3D Graphics for cartoons : Pigmentz authoring tool, Standardization Report ISO/IEC JTC1/SC29/WG11, MPEG2007/M14198, Marrakech, Morocco, January 2007.

9. M. Preda, S.M. Tran, D. Tran, I. Arsov, F. Preteux, www.3DoD.org: an MPEG-4 3D database, Standardization Report ISO/IEC JTC1/SC29/WG11, MPEG2006/M13962, Hangzhou, China, October 2006.

10. M. Preda, I. Arsov, F. Preteux, MPEG-4 3D Graphics rendering based on DirectX, Standardization Report ISO/IEC JTC1/SC29/WG11, MPEG06/13591, Klagenfurt, Austria, July 2006.

11. M. Preda, T. Laquet, W.V. Raemdonckn, I. Arsov, B. Jovanova, F. Preteux, MPEG-4 3D Graphics on mobile phone, Standardization Report ISO/IEC JTC1/SC29/WG11, MPEG06/13179, Montreux, Switzerland, April 2006.

Appendix B

BIFS Scene-Graphs

I.1 Main Menu

```
OrderedGroup {
 children [
  Background2D {  backColor 1 1 1  }
  Transform2D {
   translation 0 60
   children [
    Shape {
     appearance DEF MENU_APP Appearance {
      material Material2D {
       emissiveColor 0 0 1
       filled TRUE
      }
     }
     geometry Text {
      string ["New Game"]
      fontStyle DEF MENU_NEW FontStyle {
       justify ["MIDDLE"]
       size 26
      }
     }
    }
   ]
  }
  Transform2D {
   translation 0 20
   children [
    Shape {
     appearance USE MENU_APP
     geometry Text {
      string ["Connect"]
      fontStyle DEF MENU_CONN FontStyle {
       justify ["MIDDLE"]
       size 20
```

```
      }
     }
    }
   ]
  }
  Transform2D {
   translation 0 -20
   children [
    Shape {
     appearance USE MENU_APP
     geometry Text {
      string ["Quit"]
      fontStyle DEF MENU_QUIT FontStyle {
       justify ["MIDDLE"]
       size 20
      }
     }
    }
   ]
  }
  ...
 ]
}
```

I.2 Configuration

I.2.1 Car Selection - Images

```
OrderedGroup {
 children [
  Background2D {  backColor 1 1 1  }
  Layer2D {
   size 240 250
   children [
    Shape {
     appearance Appearance {
      texture ImageTexture { url "./car1.jpg"}
     }
     geometry Rectangle { size 150 200}
    }
   ]
  }
  Layer2D {
   children [
    Transform2D {
     translation -90 140
     children [
      Shape {
       appearance DEF MENU_APP Appearance {
        material Material2D {
         emissiveColor 0 0 1
```

```
     filled TRUE
    }
   }
   geometry Text {
    string ["<<<"]
    fontStyle DEF MENU_LEFT FontStyle {
     justify ["MIDDLE"]
     size 20
    }
   }
  }
 ]
}
Transform2D {
 translation 0 140
 children [
  Shape {
   appearance USE MENU_APP
   geometry Text {
    string ["Select"]
    fontStyle DEF MENU_SELECT FontStyle {
     justify ["MIDDLE"]
     size 26
    }
   }
  }
 ]
}
Transform2D {
 translation 90 140
 children [
  Shape {
   appearance USE MENU_APP
   geometry Text {
    string [">>>"]
    fontStyle DEF MENU_RIGHT FontStyle {
     justify ["MIDDLE"]
     size 20
    }
   }
  }
 ]
}
Transform2D {
 translation -90 -150
 children [
  Shape {
   appearance USE MENU_APP
   geometry Text {
    string ["Back"]
    fontStyle DEF MENU_BACK FontStyle {
     justify ["MIDDLE"]
```

```
         size 20
        }
       }
      }
     ]
    }
   ]
  }
  ...
 ]
}
```

I.2.2 Car Selection - 3D Model

```
OrderedGroup {
 children [
  Background2D {  backColor 1 1 1  }
  Layer3D {
   size 240 250
   navigationInfo DEF NAV NavigationInfo {type ["Examine","ANY"] }
   children [
    Inline {url "./car1.mp4"}
   ]
  }
  Layer2D {
   children [
    Transform2D {
     translation 0 140
     children [
      Shape {
       appearance DEF MENU_APP Appearance {
        material Material2D {
         emissiveColor 0 0 1
         filled TRUE
        }
       }
       geometry Text {
        string ["Select"]
        fontStyle DEF MENU_SEL FontStyle {
         justify ["MIDDLE"]
         size 26
        }
       }
      }
     ]
    }
    Transform2D {
     translation -90 -150
     children [
      Shape {
       appearance USE MENU_APP
       geometry Text {
```

```
        string ["Back"]
        fontStyle DEF MENU_BACK FontStyle {
         justify ["MIDDLE"]
         size 20
        }
       }
      }
     ]
    }
   ]
  }
  ...
 ]
}
```

I.3 Gameplay

```
OrderedGroup {
 children [
  Background2D {  backColor 0 0 0  }
  Layer3D {
   size 240 320
   navigationInfo DEF NAV NavigationInfo {type ["Examine","ANY"] }
   children [
    Inline {url "track1.mp4"}
    DEF TR_CAR1 Transform {
     translation 24 10 0
     children [
      Inline {url "car1.mp4"}
     ]
    }
    DEF TR_CAR2 Transform {
     translation 25 10 0
     children [
      Inline {url "car2.mp4"}
     ]
    }
   ]
  }
  Layer2D {
   children [
    Transform2D {
     translation 60 130
     children [
      Shape {
       appearance Appearance {
        material DEF MAT_BR Material2D {filled TRUE emissiveColor 0 1 0}
       }
       geometry Rectangle { size 20 40 }
      }
     ]
```

```
}
Transform2D {
 translation 90 130
 children [
  Shape {
   appearance Appearance {
    material DEF MAT_TI Material2D {filled TRUE emissiveColor 0 0.5 0}
   }
   geometry Rectangle { size 20 40 }
  }
 ]
}
Transform2D {
 translation -110 140
 children [
  Shape {
   appearance DEF INFO_APP Appearance {
    material Material2D {
     emissiveColor 1 1 1
     filled TRUE
    }
   }
   geometry Text {
    string ["Laps :"]
    fontStyle DEF INFO_STYL FontStyle {
     justify ["BEGIN"]
     size 20
    }
   }
  }
 ]
}
Transform2D {
 translation -50 140
 children [
  Shape {
   appearance USE INFO_APP
   geometry DEF TXT_LAPS Text {
    string ["0/3"]
    fontStyle USE INFO_STYL
   }
  }
 ]
}
Transform2D {
 translation -110 115
 children [
  Shape {
   appearance USE INFO_APP
   geometry Text {
    string ["No players : 2"]
    fontStyle USE INFO_STYL
```

```
        }
       }
      ]
     }
    ]
   }
   ...
 ]
}
```

Appendix C

MPEG-4 Player Class Diagram

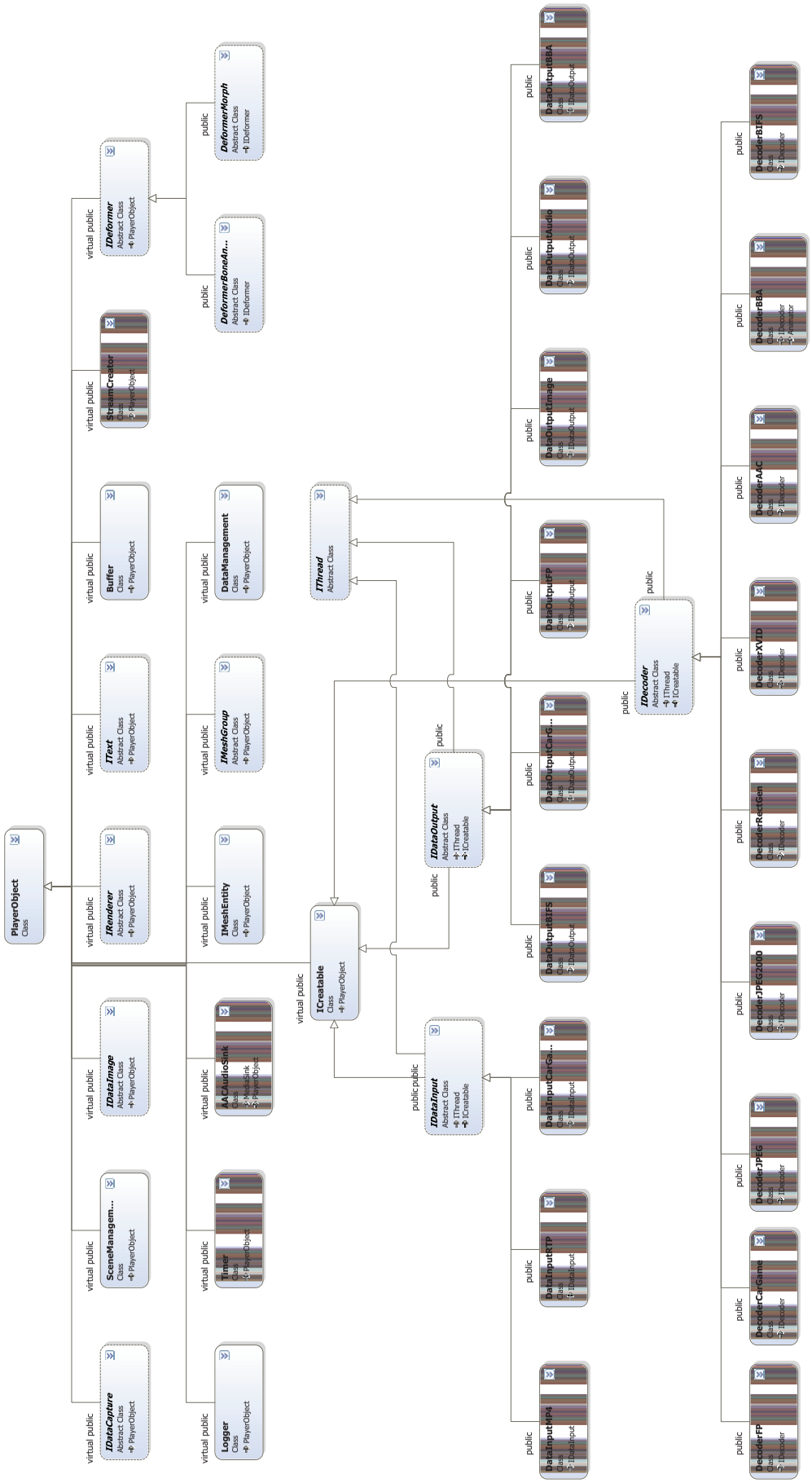


Figure C.1: MPEG-4 Player Classes and Their Dependencies

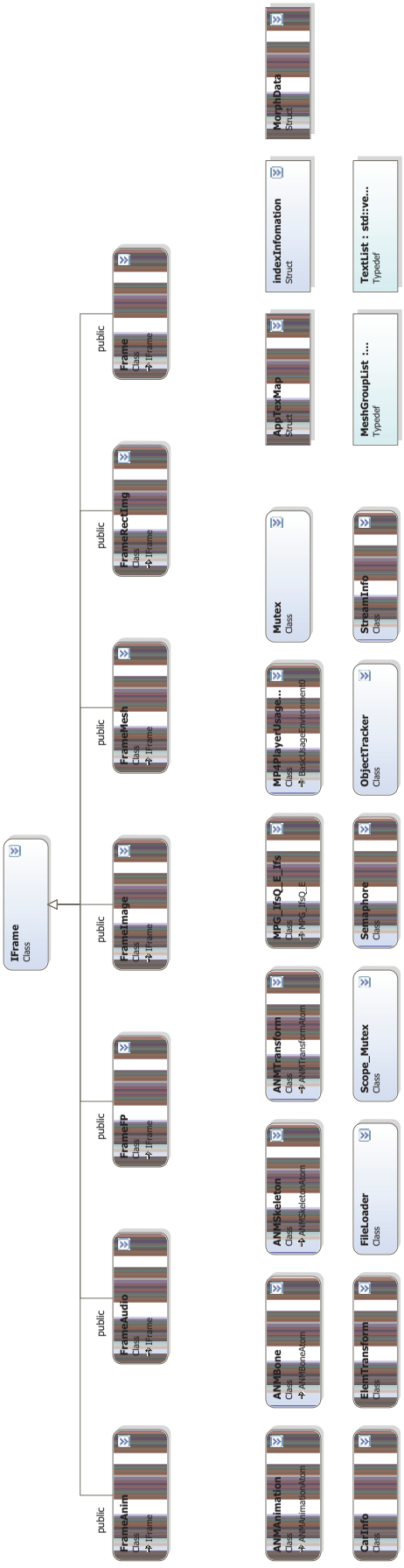


Figure C.2: MPEG-4 Player Classes and Their Dependencies