THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par *Institut National Polytechnique de Toulouse*
Discipline ou spécialité : *Informatique*

Présentée et soutenue par *Sebastien Mondet*
Le *8 juin 2009*

Titre : *Modélisation et distribution adaptatives de grandes scènes naturelles*

JURY
Stefanie Hahmann, Professeur à l'Institut de Technologie de Grenolble
Eckehard Steinbach, Professeur à la Technische Universität München
Mathias Paulin, Professeur à l'Université de Toulouse
Wei Tsang Ooi, Asistant Professeur à la National University of Singapore
Géraldine Morin, Maître de Conférences à l'Université de Toulouse
Romulus Grigoras, Maître de Conférences à l'Université de Toulouse

---

Ecole doctorale : *Mathématiques, Informatique et Telecommunications de Toulouse*
Unité de recherche : *Institut de Recherche en Informatique de Toulouse*
Directeur(s) de Thèse : *Mathias Paulin*
Rapporteurs : *Stefanie Hahmann, Eckehard Steinbach*

University Of Toulouse

ADAPTIVE MODELING AND DISTRIBUTION OF LARGE NATURAL SCENES

Sebastien Mondet

Thesis Committee:

Supervisor:	Mathias Paulin	(University of Toulouse)
Co-advisors:	Géraldine Morin	(University of Toulouse)
	Romulus Grigoras	(University of Toulouse)
	Wei Tsang Ooi	(National University of Singapore)
Examiners:	Stefanie Hahmann	(Grenoble Institute of Technology)
	Eckehard Steinbach	(Munich University of Technology)

Title: Adaptive Modeling and Distribution of Large Natural Scenes

Author: Sebastien Mondet

Advisors: Mathias Paulin, Géraldine Morin, Romulus Grigoras

Abstract:
This thesis deals with the modeling and the interactive streaming of large natural 3D scenes. We aim at providing techniques to allow the remote walkthrough of users in a natural 3D scene ensuring botanical coherency and interactivity.
First, we provide a compact and progressive representation for botanically realistic plant models. The topological structure and the geometry of the plants are represented by generalized cylinders. We provide a multiresolution compression scheme, based on standardization and instantiation, on difference-based decorrelation, and on entropy coding.
Then, we study efficient transmission of these 3D objects. The proposed packetization scheme works for any multiresolution 3D representation. We validate our packetization scheme with extensive experiments over a WAN (Wide Area Network), with and without congestion control (Datagram Congestion Control Protocol).
Finally, we address issues on streaming at the scene-level. We optimize the viewpoint culling requests on server-side by providing an adapted data-structure and we prepare the ground for our further work on scalability and deployment of distributed 3D streaming systems.

Keywords: Streaming, Plant models, Multiresolution, Progressive coding, Progressive transmission, Networked Virtual Environment

Titre : Modélisation et distribution adaptatives de grandes scènes naturelles

Auteur : Sebastien Mondet

Encadrants : Mathias Paulin, Géraldine Morin, Romulus Grigoras

Résumé :
Cette thèse traite de la modélisation et la diffusion de grandes scènes 3D naturelles. Nous visons à fournir des techniques pour permettre à des utilisateurs de naviguer à distance dans une scène 3D naturelle, tout en assurant la cohérence botanique et l'interactivité.
Tout d'abord, nous fournissons une technique de compression multi-résolution, fondée sur la normalisation, l'instanciation, la décorrélation, et sur le codage entropique des informations géometriques pour des modèles de plantes.
Ensuite, nous étudions la transmission efficace de ces objets 3D. L'algorithme de paquétisation proposé fonctionne pour la plupart des représentations multi-résolution d'objet 3D. Nous validons les techniques de paquétisation par des expériences sur un WAN (Wide Area Network), avec et sans contrôle de congestion (Datagram Congestion Control Protocol).
Enfin, nous abordons les questions du streaming au niveau de la scène. Nous optimisons le traitement des requêtes du côté serveur en fournissant une structure de données adaptée et nous préparons le terrain pour nos travaux futurs sur l'évolutivité et le déploiement de systèmes distribués de streaming 3D.

Mots-clés : Streaming, Modèles de plantes, Multirésolution, Codage progressif, Transmission progressive, Environnement Virtuel Distribué

Titulo: Modelización y distribución adaptables de grandes escenas naturales

Autor: Sebastien Mondet

Supervisores: Mathias Paulin, Géraldine Morin, Romulus Grigoras

Resumen:
Esta tesis se refiere a la modelización y la distribución de escenas 3D interactivas naturales. Nuestro objetivo es proporcionar la tecnología para permitir a los usuarios navegar en una escena 3D natural a distancia al tiempo que se garantiza la coherencia botánica y la interactividad.
En primer lugar, se proporciona una técnica de compresión multiresolución, sobre la base de la normalización, la instanciación, la decorrelación, y la codificación entropica de la información geométrica de modelos de plantas.
A continuación, se estudia la transmisión eficaz de estos objetos 3D. El algoritmo de paquetización que proponemos funciona con la mayoría de las representaciones multiresolución de objetos 3D. Validamos las técnicas paquetización por experimentos en una Red de Área Amplia (WAN), con y sin control de la congestión (Datagram Congestion Control Protocol).
Por último, se aborda las cuestiones del streaming al nivel de la escena. Optimizamos el procesamiento de consultas en el lado del servidor, proporcionando una estructura de datos adaptada y preparamos el terreno para nuestro trabajo futuro sobre la escalabilidad y el despliegue de los sistemas distribuidos de streaming 3D.

Palabras clave: Streaming, Modelos de plantas, multiresolución, codificación entropíca, transmisión progresiva, Entorno Virtual Distribuido

大规模自然场景的自适应建模和分发

本论文研究大规模三维自然场景的建模和交互性流传输，目标是使用户能够在三维自然场景中远程漫游，并且保证交互性和植物的眞实性（植物学意义上与眞实植物保持一致）。

首先，本文为（植物学意义上严格）眞实的植物模型提供一个紧凑和渐进(progressive)的表现方式。
此模型中，植物的拓扑结构和几何特性以通用柱体来表示。本文提供了一个基于标准化和实例化， 差值解相关， 和熵编码的多分辨率的压缩体系。

然后，本文研究了上述三维物体的高效传输。文中给出的打包体系适用于任何多分辨率的三维表现方式。本文用广域网上大量的实验（实验中既用了有拥塞控制的协议，也用了无拥塞控制的协议）来验证了这个打包机制。

最后，本文解决了场景层面上的流传输中的一些问题。文中提供了一个改进的数据结构来优化服务器端的视点挑选(culling)操作这为我们将来研究分布式三维流媒体传输系统的可扩展性和部署问题打下了基础。

Titlu: Adaptarea modelizării și distribuției scenelor naturale de mari dimensiuni

Autor: Sebastien Mondet

Indrumatori: Mathias Paulin, Géraldine Morin, Romulus Grigoras

Rezumat:
Teza tratează modelizarea și difuzarea scenelor 3D naturale de mari dimensiuni. Aceate lucrare propune tehnici ce permit utilizatorilor să navigheze de la distanța într-o scenă 3D naturală, asigurând în același timp coerența botanica și o bună interactivitate.
Într-o primă etapă, prezentam o tehnică de compresie multirezoluție, bazată pe normalizarea, instanțierea, decorelarea și codajul entropic al informațiilor geometrice privind modelele de plante.
În continuare, studiem transmisia eficace a acestor obiecte tridimensionale. Algoritmul de pachetizare propus funcționează pentru majoritatea reprezentarilor 3D multirezoluție ale unui obiect. Astfel, validăm tehnicile de pachetizare prin experimente realizate în cadrul unei retele WAN (Wide Area Network) cu și fără control de congestie (Datagram Congestion Control Protocol).
Partea finală abordează problema streaming-ului la nivelul scenei. Prezentăm așadar optimizarea tratamentului cererilor serverului prin furnizarea unei structuri de date adaptate și pregătim terenul pentru dezvoltări ulterioare vizând scalabilitatea și implementarea sistemelor distribuite de streaming.

Cuvinte cheie: Streaming, Modele de plante, Multirezoluție, Codaj progresiv, Transmisie progresivă, Medii Virtuale Distribuite

Titel: Adaptive Modellierung und Verteilung grosser natürlicher Szenen

Autor: Sebastien Mondet

Doktorväter: Mathias Paulin, Géraldine Morin, Romulus Grigoras

Zusammenfassung:
Diese Dissertation beschäftigt sich mit der Modellierung und dem interaktiven Streaming von großen natürlichen 3D-Szenen. Unser Ziel ist es, Methodiken zu erarbeiten, die es einem ueber ein Netzwerk angebundenen Benutzer ermoeglichen, durch eine natuerlich Szene zu navigieren. Besonderes Augenmerk liegt dabei auf der Erhaltung der botanischen Kohärenz und der Interaktivität.
Unser erstes Ziel ist es, eine kompakte und stufenweise Darstellung botanisch realistischer Modelle von Pflanzen zu erarbeiten. Die topologische Struktur und die Geometrie der Pflanzen werden durch verallgemeinerte Zylinder repraesentiert. Wir erarbeiten ein multiresolution Komprimierungsschema, das auf Standardisierung und Instanziierung, auf Differenzkorrelation und auf Entropiekodierung aufbaut.
Anschliessend untersuchen wir die effiziente Übertragung dieser 3D-Objekte. Das vorgeschlagene Packetierungsschema kann auf alle multiresolution 3D-Darstellungen angewendet werden. Wir validieren unser Packetierungsschema in umfangreichen Experimenten über ein WAN (Wide Area Network), mit und ohne Verstopfungskontrolle (Datagram Congestion Control Protocol).
Abschliessend befassen wir uns mit Fragen des Streamings auf der Ebene der Szene selbst. Wir optimieren die Anfragen auf Serverseite basierend auf Sichtbarkeitsanalyse in einer angepassten Datenstruktur.
Diese Dissertation legt die Basis fuer weitere Arbeiten im Bereich der Skalierbarkeit und Verfuegbarkeit im verteilten 3D Streaming.

Schlagworte: Streaming, Pflanzenmodelle, Multiresolution, Stufenweise Darstellung, Stufenweise Übertragung, Vernetzte Virtuelle Umgebung

Contents

Chapter 1

Introduction

Contents

Availability of high bandwidth Internet connections at home and powerful graphics hardware on commodity PCs have increased the popularity of networked virtual environment (NVE) applications. NVEs are one of a few truly *multi-media* applications that involve many media types: 3D models, animation, images, audio, and video. These media data are typically stored on a server, collectively describing a virtual environment. A client connects to the server to navigate through the environment, requesting a subset of the media data based on its current viewpoint. The server transmits the requested media data to the client, which receives it and creates a partial/local 3D scene that is further rendered into a virtual environment at the client.

The multimedia research community has made much progress on audio and video transmissions, enabling high quality audio communications and video streaming within the NVE. The quality of 3D objects in NVEs, however, is still primitive and not realistic in general. Simplified models or image-based representations are commonly used in NVE to reduce both computational and bandwidth requirements. While Moore's Law and advances in GPU (Graphics Processing Unit) technology have made concerns on computational requirements less relevant, network bandwidth still remains a major bottleneck. For instance, current generation

Figure 1.1 – Photography of a Bavarian landscape and screenshot of the *Natsim Visualization Tool*.

of GPU is capable of rendering in real-time the Stanford's Thai Statue model with 10 millions triangles, but the model, with a size of 122 MB after compression, still needs 1.6 minutes to download even on a fast 10 Mbps link. The latency induced by completely downloading such an object during a client navigation is prohibitive for interactive use. Thus, to enable realistic, high resolution 3D object in NVE, it is not acceptable to render a 3D object only after it is completely received.

Among the data that one would aim to represent in a NVE, the world we live in is certainly the most obvious. It surrounds us and any realistic virtual representation should reproduce it faithfully. On one hand, the botanic, biologic and physics communities acquire and store huge data sets representing each single natural entity with a dedicated model. On the other hand, the user community is willing to smoothly navigate in realistic virtual environments with complex plants (trees, forests, meadows), watercourses (rivers, rivulets, waterfalls) and atmospheric phenomena (clouds, mist, fog).

In the present work, we study the interactive streaming in large natural scenes. The final goal is to provide a system able to allow a set of independent users to interactively walk through a *remote* large natural scene. The 3D scene may be stored on one or more servers. The clients may be using different devices (desktop computers, smart phones). The network lying in between is a best-effort IP network (e.g. the Internet).

This document presents our work performed from October 2006 to March 2009 within a PhD studentship. This first chapter of the thesis aims at giving an overview of the context of the thesis (section 1.1), and a presentation of the research topic (section 1.2).

1.1 The Context

This work has been supervised by Mathias Paulin, Géraldine Morin and Romulus Grigoras at Toulouse's Computer Science Laboratory (IRIT) from the University

of Toulouse within the VORTEX research group (Visual Objects from Reality To EXpression). Wei Tsang Ooi from National University of Singapore has jointly supervised this thesis, especially during a three-months internship at the School of Computing of the NUS, from June to August 2008.

The work done during this thesis preparation has been mostly funded by the *NatSim* project (section 1.1.1) and is a follow-up to a previous study we performed in 2005 (section 1.1.2).

1.1.1 The Natsim Project

The *Nature Simulation Project*[1], funded by the French National Research Agency (ANR) had the code name: `05-MMSA-0004-01`. It aimed at studying and providing tools for modeling, representing, and transmitting natural scenes from a simultaneous *computer graphics* and *multimedia* point of view.

Despite a growing interest, this emerging research topic had received little attention. Hence, the project focussed on the models, the evolution, the adaptive transmission, and the visualization, but also on the composition of several natural entities in a complex virtual environment.

Natsim brought together a multidisciplinary consortium of participants having complementary expertise covering a wide range of skills:

- IRIT laboratory, through the VORTEX research team, had acquired recognized knowhow on point-based graphics, multiresolution, visualization and streaming (c.f. section 1.1.2) but these works had only been applied to static scenes;
- EVASION project was a well established research team in natural phenomena modeling, visualization, and simulation;
- CIRAD (formerly AMAP) was internationally recognized in the domain of plant modeling. The research groups *Virtual Plants* and *Stand and Landscape* brought into the project their botanical and computer science expertize in plant architecture analysis, modeling and simulation.
- IPARLA team had acquired a solid experience on point-based graphics and hardware dependent visualization (from smart phones to reality center);
- LIAMA, through the GreenLab project, had been developing research on plant, crop and landscape visualization.

Five "work-packages" were the basis of NatSim:

1. *Multi-model representation* worked on multi-resolution, multi-model representation of natural scenes;
2. *Acquisition, editing and modeling* studied the *input* part of the pipe-line;

[1] c.f. www.irit.fr/Natsim

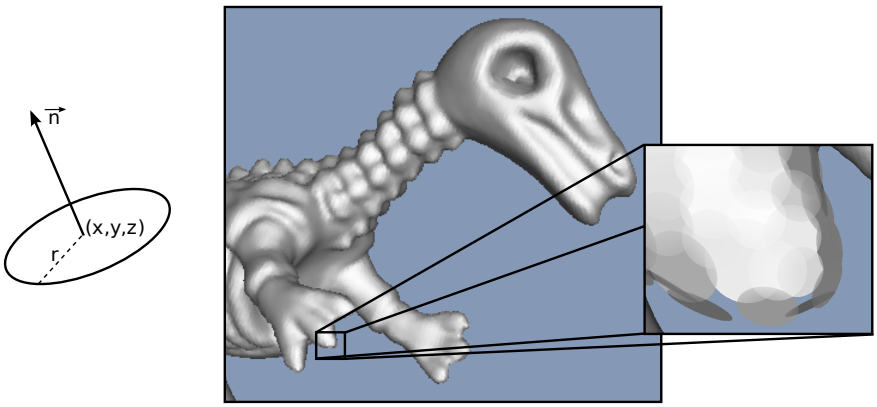

Figure 1.2 – The structure of a Splat (Point-based surface base element), and an example of rendering.

3. *Rendering and lighting simulation* provided real-time rendering of natural scenes (c.f. figure 1.1);

4. *Animation and simulation* focused on the dynamic aspect;

5. *Streaming*, our work-package, has been working on the streaming of large natural scenes.

The *Streaming* work-package was intended to provide a distribution framework for remote visualization, while participating in the *Multi-model representation* work-package. This double context has for example allowed us to initiate collaboration with the team Virtual Plants from the CIRAD (Frédéric Boudon, Christophe Pradal, and Christophe Godin).

1.1.2 Streaming of Point-Based 3D Scenes

This thesis follows previous work we have done on *streaming of point-based 3D scenes*. A master thesis has been prepared during the period from February to June 2005. This work is described, in French, in "Mise en ligne d'environnements 3D vastes échelonnables : adaptation aux ressources et à la navigation" (Mondet, 2005)[2] and, in [MMG05].

This previous work was based on a client-server architecture providing interactive walk-through of distant 3D scenes modeled as point-sampled geometry.

Point-based geometry, even if presented long ago by Marc Levoy and Turner Whitted (c.f. [LW85]), has gained recently a lot of attention from the *Computer Graphics* community (c.f. [RL00, KB04, Pau03, BSK04] and figure 1.2). We had chosen point based geometry because it is inherently much more fault tolerant than meshes. To manage the points on server-side, our system was based on a Kd-Tree

[2] Available online: sebmondet.ifrance.com/RapportS3DSebastienMONDET.pdf

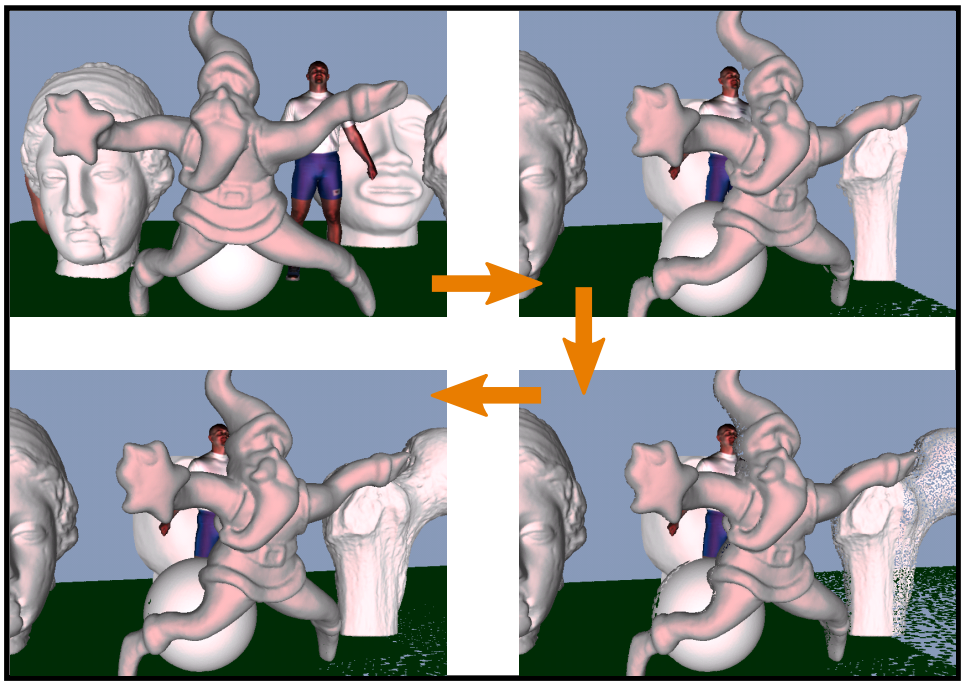

Figure 1.3 – Screenshot of the point-based streaming client with viewpoint adaptation.

structure representing the scene (for a complete presentation of Kd-Trees please c.f. [BKOS97]). This structure allowed us to process, on the server, viewpoint requests on the geometry. The server selected the visible part of the scene to send to the client (shown in figure 1.3). Regularly the client sent its viewpoint to server (if changed). The client application was based on PointShop3D's rendering engine. PointShop3D is a modeling suite for point-based 3D models, developed at the EPFL [ZPKG02].

On the network, three different underlying protocols were studied:

- **HTTP:** the system used the `apache` web-server and a custom CGI-application replied to the client's requests (the viewpoint was encoded in the URL);
- **TCP:** a custom TCP server was much more reactive and much less resource-greedy;
- **DCCP:** the protocol was still experimental: IETF[3] only provided drafts of the RFC[4], and only one user-space and event-based implementation was available.

[3] Internet Engineering Task Force
[4] Request For Comments

At the end of the master thesis, both the HTTP-based and the TCP-based were working C++ implementations of the client-server system. The DCCP-based one didn't reach a stable state before the end of the master work.

This previous work was our first experience with 3D streaming.

1.2 Streaming of Large Natural 3D Scenes

As the NatSim project defined it, our mission is to make available 3D natural scenes for remote visualization. We aim at allowing distant users to experience interactive walk-through over the internet. Client may have heterogeneous resources, and network conditions may be variable.

Natural scenes have special models and structure. Specific models for plants [Blo85], for waterways [YNBH09] or clouds [BNM+08] have been proposed. Our goal is to keep as much as possible the botanical and physical coherency of the scene.

Next section gives some examples of application fields of our topic (section 1.2.1). Then, we provide detailed overview of the problem of 3D streaming applied to natural scenes (section 1.2.2).

1.2.1 Target Applications

The first application that motivates our work comes directly from the NatSim project. Joint Computer Science and Botanical research are setting huge datasets resulting from the simulation of the growth of forests and natural environments. The work-package in charge of the real-time rendering of natural scenes provides visualization tools for these simulated and botanically realistic environments. The specialized 3D streaming lies in between. The visualization tool should be a client application for the remote interactive visit of these scenes. The approximations needed should keep the botanical realism of the simulations.

A few botanical gardens around the world already provide some image-based online visits. For instance, the website "Explore Kew Gardens"[5] proposes an "Official Virtual Tour" of the *Royal Botanic Gardens* at Kew, near London. *Explore Kew Gardens* makes available 360-degrees image-based panoramas, small movies, viewpoint narrations, maps, and text. The website is based on Flash and Java technologies. A totally interactive immersion in a virtual botanic garden would be a good enhancement. Additional botanic information could be provided and, moreover, animated growth of the featured plants could be shown.

Another application of the *botanical coherency* are the *nature-oriented* educational games. Such online games would, for example, propose a remote "treasure hunt" based on botanical and biological riddles.

Finally, adaptation to heterogeneous user devices could provide tools for mobile workers like landscape designers using *Personal Digital Assistants* (PDA) or

[5] c.f. www.explore-kew-gardens.net

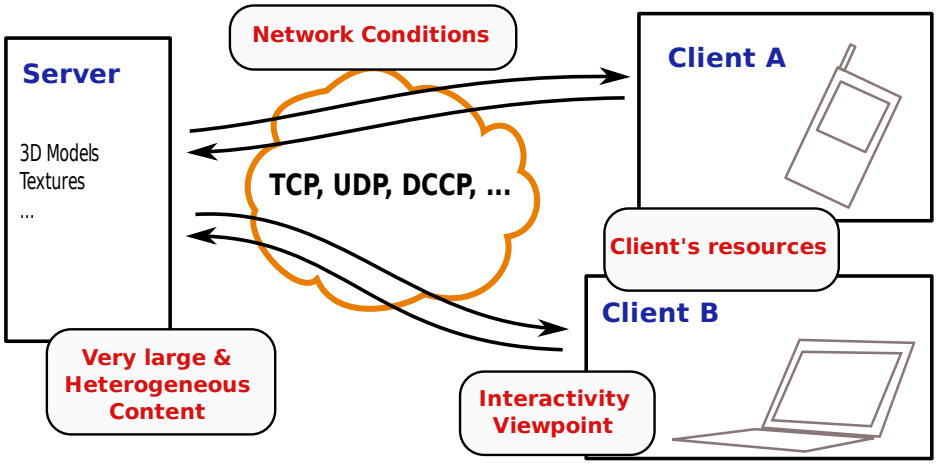

Figure 1.4 – Main idea of a client-server system with issues raised by 3D streaming

Smart Phones. A rendering of the potential created landscape could be provided on site. Moreover it would be interesting to put in relation the virtual viewpoint (the *camera* of the rendering engine) and the actual position of the device. Such a "Mobile Future Viewer" would use for example, the GPS (Global Positioning System) chip, the accelerometer and the camera which are embedded in modern PDAs and phones.

1.2.2 Issues and General Solutions

1.2.2.1 Four Main Issues

Figure 1.4 presents a simple client-server 3D streaming system. Our use case is: clients having different computing devices connect to the server to visualize a 3D scene. The server transmits it progressively. The main issues raised are shown on the figure. In the following, we detail these constraints.

Very Large Content

Natural (or non-natural) scenes may be composed of hundreds of 3D objects, and other materials (like textures). Even a single object can be large in terms of storage. For example, the geometry of the statue of David, from the Digital Michelangelo Project, represented as a mesh, consists in 2 billion polygons; even with (lossless) compression the total file size is 32 GB. Downloading completely such a model before visualizing is already completely prohibitive. Hence for 3D scenes we need accurate and progressive streaming. The sent content must not be useless (e.g. not visible) and incoming data must be decodable as soon as possible, portions of the data should lead to visual approximations of the 3D content.

Heterogeneous Client Devices

Nowadays clients of internet-based distributed applications are expected to have very heterogeneous devices. Devices like Personal Digital Assistants (PDAs) or Smart Phones are now able to render simplified 3D scenes. Graphics-hardware manufacturers now provide embedded chips with rendering capabilities, such as NVidia's GoForce chipset which provides acceleration for 3D rendering since the "4500" model. On the other hand, small devices may even not have hardware-based floating-point arithmetic. For example the ARM architecture (Acorn RISC Machine), which is the most widely used on mobile platforms, is a 32-bit RISC CPU which only provides floating-point arithmetic as optional co-processing units with certain versions (e.g. the VFP extension). To handle these limitations on the rendering part, the Khronos Group has defined a specialized standard API for 2D and 3D graphics on embedded systems: *OpenGL ES*[6] as a subset of the "desktop" OpenGL API and some extensions providing *easy-to-use* fixed point arithmetic. Hence, we must consider that 3D streaming clients may have a wide range of devices, with different 3D rendering capabilities: from simple smart phones to heavy desktop computers equipped with powerful graphics cards.

Interactivity

Whatever device he is using, the client must be allowed to walk through the scene following any random path. It means that client's viewpoint is always changing and is mostly unpredictable. That is the main difference between 3D and video streaming. For 3D, even if the viewer's position can be considered "slowly continuous" (i.e. if we consider *jumps in space* relatively rare), a small rotation of the viewpoint may imply "kilometers" for the visible horizon. The set of visible objects can change dramatically. On the other hand, the viewpoint of a video viewer has only one dimension: time. For most usages of video streaming, the viewer will see the video continuously, even if enhanced streaming systems handle the forward and backward jumps in the video sequence as exceptional cases (c.f. [LGS+00]).

Network Conditions

The common *unsafe* ground of all distributed systems is the network. We consider the internet network of networks as our base. This means that we must handle variable conditions and no guaranteed quality of service. Bandwidth is variable and represents often a bottleneck for the application. Losses and disordering of packets are common too. A streaming system must be able to lower its requirements to provide acceptable content to the viewer, even when network conditions worsen.

1.2.2.2 One Keyword: Adaptation

There is a growing mismatch between the use of more and richer multimedia content and the need to access it in an ubiquitous manner. Universal Media Access

[6] c.f. www.khronos.org/opengles

(UMA, c.f. [MSL99]) states it nicely: provide access to advanced multimedia content anywhere, anytime, and with any kind of device. Today we can only design a distributed multimedia application or produce multimedia content with heterogeneity and context awareness in mind. Heterogeneity means that we should take into account constraints that are known beforehand (devices, networks, user interactivity requirements). Context awareness means that a multimedia application needs to adapt, at run-time, to an evolving context (e.g. variable network conditions, dynamic user load on servers, geographic position of a user etc.).

Adaptation to predictable or unpredictable constraints/factors is therefore paramount in modern distributed multimedia systems. We are naturally in line with this approach, since we consider adaptation issues at every level in our 3D streaming system. We detail now the different aspects of *Adaptation* solutions for our problem.

Compression

The first *solution* is to transform the data, i.e. adapt the 3D scene to our needs. Since the content can be very large, and it must fit in a relatively thin pipe: the network bandwidth. Therefore our first goal is to represent the content as compactly as possible. Compression techniques must be used, statically or on-the-fly, to transmit smaller data over the networked channels.

Progressiveness

Another pre-processing treatment we can provide for the content is the transformation to a progressive (or *multiresolution*) representation. By progressiveness we mean that the content must be organised to be partially decodable. First portions of a progressive data-stream lead to the decoding of a coarse approximation of the modeled object (i.e. *low-resolutions*), and the following parts improve the quality of the approximation (i.e. *higher resolutions*).

Progressiveness is a key tool for multimedia adaptation. When streaming very large content, one can provide quickly a coarse resolution whose rendered representation gives visual feedback to user, while waiting for the refinements. Moreover, coarse approximations of a given objects may allow the user continue the *visit* if the object is not of his interest; higher resolutions may not need to be transmitted at all. On the other hand, when clients use different devices in terms of memory and rendering capabilities, a streaming system must adapt the content to the device; multiresolution encodings allow to (*dynamically*) degrade the transmitted model to adapt them to the resources of the rendering device. Finally, progressiveness of the content allow a streaming system to adapt to the variability of network conditions, for example by lowering the resolution requirements when the network is congested.

Efficient packetization

From a multimedia application point of view, a transmission over the network is either a stream, i.e. an ordered sequence of bytes, if using TCP-like protocols, or a succession of unordered and potentially lost packets if using UDP-like protocols. In the former case, the peer's operating systems hide the losses and disorderings of the underlying network to the application, but with a performance hit. In the latter case, the application has the flexibility of optimizing its performance but must manage the losses and reorderings induced by itself. Therefore the application faces the problem of the adaptation of the transmission scheme to the loss rate of the network. Actually *packing* pieces of data in packets which have a predefined maximal size taking into account the network conditions is an adaptation problem for multimedia applications. We must hence provide efficient packetization scheme, which is aware of the characteristics of the content and of the network conditions.

Pre-fetching

Interactivity, and hence the variations of the viewpoint, do not condemn 3D streaming systems designer to implement only *reactive* schemes. Pre-fetching is a common *proactive* adaptation solution in multimedia systems. It consists in transmitting *not-yet-visible* data in advance to take profit from network conditions i.e. when the bandwidth is high.

Multi-model representations

Finally, another adaptation scheme is the *multi-modality* of 3D objects. The idea is to have various representations of the same object, and adapt their use to the network conditions, the user's viewpoint or its device's capabilities. For example, a plant in a natural scene can be represented (and transmitted) as a high resolution model when viewer is inspecting it closely. But the system can choose to send only an image-based low-resolution representation (called *billboard*) when the viewer is far or when its rendering capabilities do not allow its device to display enough geometry.

1.3 Thesis Outline

The work exposed in this document aims at contributing to 3D streaming of natural scenes by proposing adaptation schemes following the part of the ideas described above (section 1.2.2.2). In the next chapter (2), we develop our study on a *compressed* and *progressive* representation of plant models which are among the most *important* objects in 3D natural scenes. Then in chapter 3, we present a *packetization* method adapted to 3D multiresolution content, and we apply it to the previous progressive model for plants. We present our work on large *scenes*, about adaptation of the system to the interactivity of the client's viewpoint, and the 3D streaming test-bed we have designed (chapter 4). The chapter 5 states about the raised

issues and the learned lessons regarding the practical and implementation aspects of the work. Finally, the chapter 6 concludes and gives global perspectives about the streaming of wide natural scenes.

Chapter 2

Progressive Representation of Plant Models

Contents

In this chapter, we propose a progressive compression scheme for plants based on *generalized cylinders*. This representation, by its multiresolution aspect, is *streaming friendly*. It allows us to packetize, transmit and render progressively plants with an increasing quality.

2.1 Relevance and Specificities of Plant Models

Realistic modeling of plants is crucial in NVE applications such as virtual forests or virtual botanical gardens, where users are (or *will be*) expected to inspect a plant closely and possibly interact with the plants they have accessed remotely.

However, realistic and detailed plant models can require up to hundreds of thousands of primitives if modeled with classical polygonal surfaces. Remolar et al., in [RCB+02], estimated that a plant generated by XFrog, a well known plant modeling platform[1], can consist of 50,000 polygons only to represent the branches. The plants can have 20,000 or more leaves, which themselves consist of polygons. Neubert et al., in [NFD07], reported the plant models that they used consist of up to 555,000 polygons. These numbers are for a single plant. In natural scenes, such as forests, one would expect the scene to contain a very large number of plants. The size of these plants motivates the need to stream progressively, rather than to wait until the complete plant model is received before being displayed. Progressiveness is motivated by performance constraints: the network bandwidth, but also the in-memory size, the distance of the plant to the viewpoint, etc.

Progressive representation for generic 3D objects are well studied. For instance, multiresolution coding of triangle meshes (c.f. [Hop96, AD01, AG05, Tau99]), point-based surfaces (c.f. [Pau03, RL00, KB04]) or hybrid representations (c.f. [CN01]) are all progressive coding schemes for 3D objects. However, these representations are not suitable for plants due to the topology structure of the branches. For example, with progressive meshes, it is difficult to remove triangles above a certain level, and as a result, representation of plants by progressive meshes does not give satisfactory results (c.f. [RCB+02]).

Figure 2.1 illustrates that simplification of a mesh tree does not preserve the topology, in particular the connectivity, of the tree. Hence, progressive representations suited to the topology of plants are needed.

[1] http://www.greenworks.de

Figure 2.1 – Mesh simplification on the *Walnut* model (section 2.3.1.2). The original model consists of 278 632 triangles, here are presented simplified models consisting of 0.1% (left) and 0.2% (right) of the original model.

Therefore, our aim is to provide a progressive and compressed representation for plants, that preserves their *botanical coherency*. We want to ensure the connectivity between branches at each stage of decoding and, if possible, the realism of the shape of the branches regarding the plant specie.

2.2 Representing and Modeling Plants

Previous work has focussed on how to accurately model a plant (c.f. [RCB+02, Blo85, PMKL01, PL90, NFD07]) or how to easily create a plant within the virtual environment as proposed, for example, by the *Dryad* project[2].

Plant geometry is particularly complex and thus motivated a variety of representations dedicated to its specific needs (c.f. [DL05, BMG06]). Branches and foliage are usually treated separately.

2.2.1 Procedural Modeling: L-Systems

From a modeling point of view, a well-known custom modeling scheme for plants are the *L-Systems* (c.f. figure 2.2). L-systems were introduced and developed in 1968 by the Hungarian theoretical biologist and botanist from the University of Utrecht, Aristid Lindenmayer (1925–1989). An L-system consists in a string representation of the branching structure coupled with a formal grammar (c.f. Prusinkiewicz and Lindenmayer's book: [PL90]). The rewriting rules of the grammar simulate the growth of the plant. The topology and the geometry of the plant are given by a LOGO-style turtle that interprets the symbols of the string as geometric commands (c.f. [Pru86, FKMP03]). We note that, in this system, the ge-

[2]Dryad: http://dryad.stanford.edu

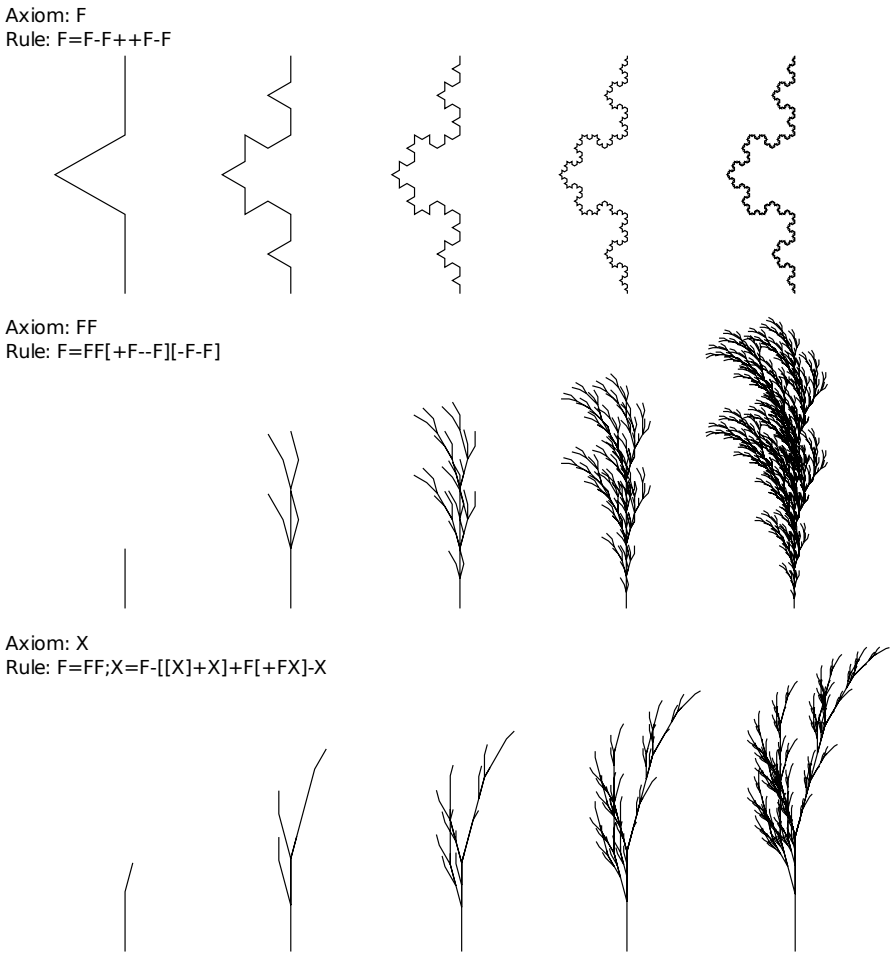

Figure 2.2 – Examples of Basic 2D L-Systems (*Generated with Inkscape*).

ometry of a symbol is built according to the geometry of previous elements, and leaves are instances at different places of the same geometric symbol.

2.2.2 Parametric and Implicit Surfaces

The previous idea has inspired a lot of work (our progressive representation stands among of them). More generally, some high level representations for branches have been proposed based on parametric (c.f. [Blo85]) or implicit surfaces (c.f. [GMW04]). They rely on a skeleton of branches which is extended with radius (given by cross sections or implicit functions). The skeleton is defined as a set of connected parametric curves. These topological structure representations have the advantage of being compact compared to more discrete representations such as mesh and provide support for animation (which is not the case of the simplified models whose connectivity is lost in figure 2.1). By default, however, they are not

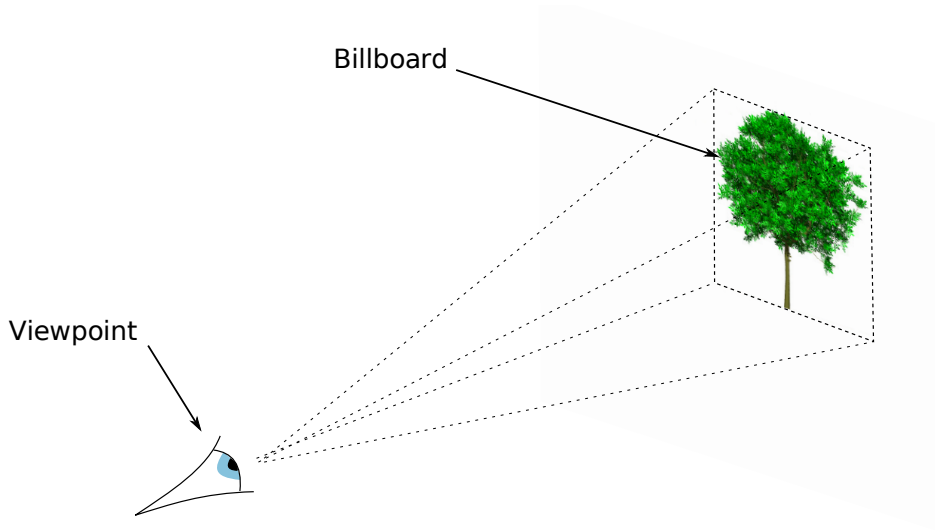

Figure 2.3 – An example of billboard: A pre-rendered image, presented as an impostor in front of the viewpoint (generally, a billboard is a texture sticked on a rectangular piece of mesh).

adapted for progressive description. Our goal in this chapter is precisely to fill this gap.

2.2.3 Models for Rendering

From a rendering point of view, some representations are based on *billboards* i.e. pre-rendered images used as impostors, c.f. [MNP01, DN04, BCF+05]; see also figure 2.3. Also representations based on points (c.f. [WP95, DCSD02]) or polygons (c.f. [RCB+02, ZBJ06]) have proposed adaptive schemes for displaying trees. These representations mainly focus on foliage (leaves) and thus can be seen as complementary to ours since they are usually complemented with polygonal representations of trunk and branches. If these representations offer some interesting results, they usually require a large amount of data, in particular with points and images. Polygonal representations on sparse geometry such as foliage are not totally convincing. These representations can be streamed with classic methods since they use classic primitives with low-level abstraction. By default, however, they seem more dedicated to static representations. Additionally, they have to be attached to a skeleton representation to support animation.

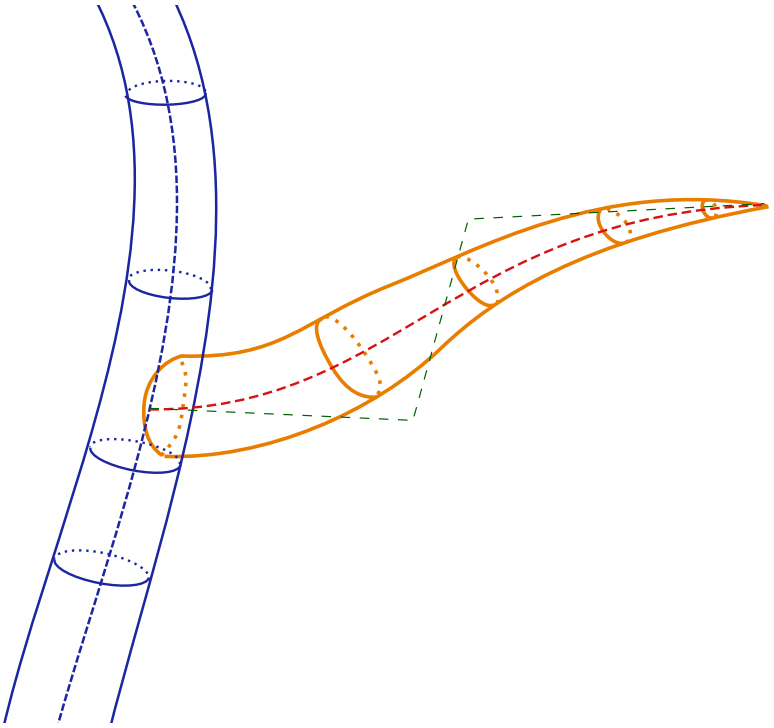

Figure 2.4 – Generalized Cylinder with Radius along the branch.

2.2.4 Generalized Cylinders

On the other hand, *Generalized Cylinders* are a representation focusing on the branching structure of the plant, i.e. "skeletal-based" (c.f. [Blo85]). A branch is represented by an axis curve, a parametric curve defined by a set of control points, and parameters defined along the branches (c.f. figure 2.4). Such generic high level representation can then be displayed as generalized cylinders (c.f. [Blo85, PMKL01]) or implicit surface (c.f. [GMW04]) and is much more compact than a mesh representation. For example, the *Walnut* (presented in section 2.3.1.2) at full resolution only requires about 10 772 control points using generalized cylinders compared to 278 632 triangles using a mesh model. Recent work has also studied the real-time rendering of generalized cylinders using modern graphics hardware (c.f. [GM03, BW05]). Additionally, this representation, which is based on a *skeleton* structure, can possibly be extended with kinetic informations for use in animation. The branches are organized inside a n-ary tree data structure modeling the structure of the plant. We call such a data structure a *n-tree*, to avoid confusion with the concrete plant object we are actually modeling. This representation has been chosen as a starting point for our work (see also [Bou04]).

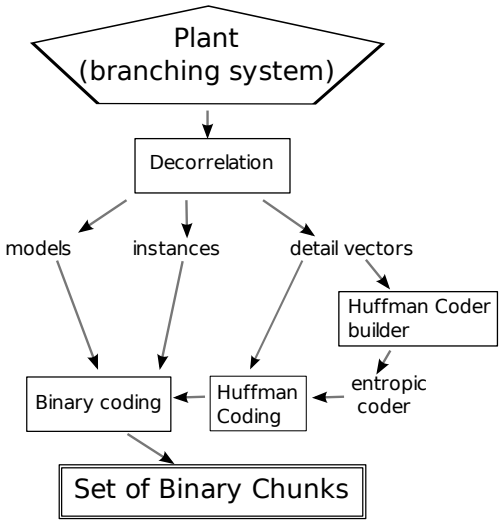

Figure 2.5 – The encoding process for a model based on skeletal representation.

2.3 A Progressive Representation of Plant Models

In this section, we detail the development of our streaming-friendly representation and coding of plant models based on generalized cylinders. This work has been first published in the paper [MCM+08] and then extended in [MCM+09].

Figure 2.5 outlines the steps from encoding to streaming of our representation, and guide the presentation of this section. Our starting point is a natural scene using plant models based on precise skeletal representation (section 2.3.1). This representation served as the basis for our proposed compressed, progressive representation that decorrelates information into three components called branch models, instances, and detail vectors (section 2.3.2). The detail vectors are compressed with entropy coding and other pieces of data are efficiently coded (section 2.3.3), resulting in a set of binary chunks. Finally, each chunk is assigned an importance value, which is then used for scheduling the chunks in the progressive decoding stream (section 2.3.4).

2.3.1 Input Data

As a starting point, in this section, we aim at providing a good understanding of what we consider as our input data. First, we describe precisely the generalized cylinder with a *data-structure viewpoint* (section 2.3.1.1). And then we present the actual plant models we have used for testing, evaluating and experimenting the methods presented in this chapter (section 2.3.1.2).

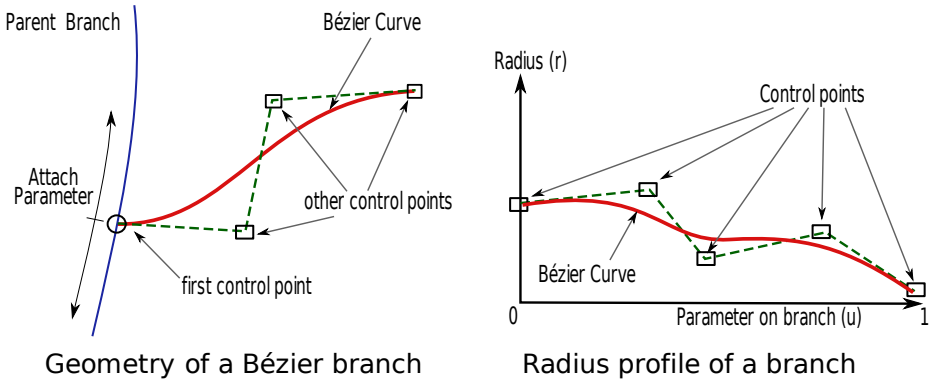

Figure 2.6 – On the left, the Bézier curve representing a branch with its attachment parameter (u) on its parent branch. On the right, the Bézier curve representing the radius along a branch (i.e. $(u, r) \in [0, 1] \times [0, \infty]$).

2.3.1.1 Bézier-Based Generalized Cylinders

Our representation focuses on the branching structure of a plant and is thus based on a skeletal representation. Each branch is a generalized cylinder: an axis curve, which is a 3D Bézier curve defined by its control points[3] and *axial parameters* such as the radius, color or texture coordinates modeled as Bézier curves along the branch. In practice, we use for now only radii as *axial parameters*, defined by 2D Bézier curves. As presented in section 2.2, we render and display this high level representation as 3D generalized cylinders (c.f. [Blo85, PMKL01]).

The branches are organized inside an n-ary tree data structure giving the structure of the plant. The root of the n-tree is the trunk of the plant and branches borne by the trunk are the n-tree children of this trunk. Each child branch contains a *attachment parameter* ($u \in [0,1]$) giving the position of the attachment point on its bearing parent branch (as in [PMKL01]). The parameter u defines the first control point of the Bézier curve of the child branch. The remaining control points are encoded in the child branch by their three coordinates in space.

Axial parameters are also defined thanks to the attachment parameter u. We take the example of the radius but the scheme can be adapted to textures or colors. The case of the radius of the branch illustrates how attributes along the branch are coded. A radius is defined as a positive real value along the branch. To model it as a smooth function along the branch, we represent its values as a series of control points (u_i, r_i) of a Bézier curve of degree m, where $(u_i)_{i=0\ldots m}$ is an increasing sequence in the interval $[0,1]$ that defines the location of the branch, and $(r_i)_{i=0\ldots m}$ characterizes the radius for the corresponding given location. Note that the degree of the radius curve is not related to the degree of its bearing branch.

Figure 2.6 summarizes the structure of the model.

[3] For extensive definition and description of Bézier curves please refer to [Far02]

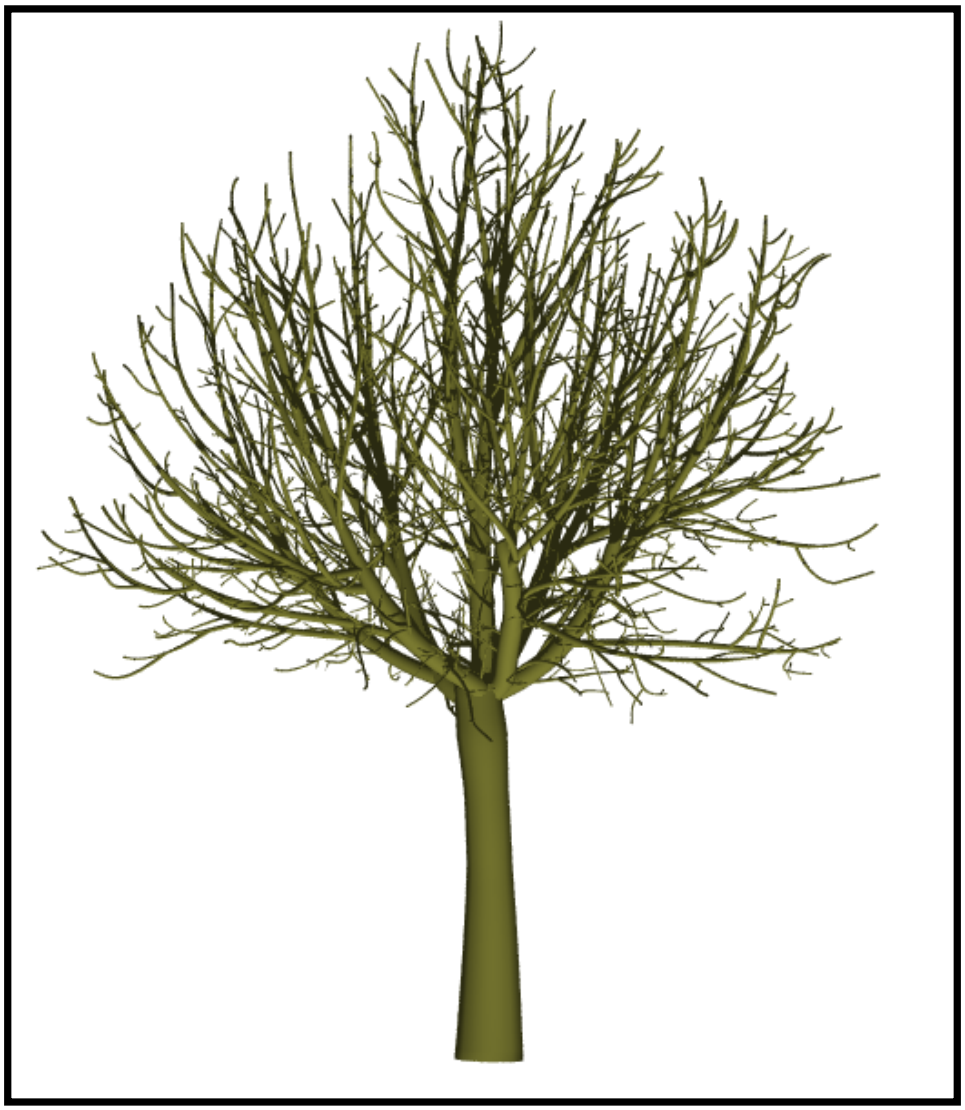

Figure 2.7 – The Walnut model (digitized by Sinoquet et al. c.f. [SRG97])

2.3.1.2 Our Plant Models

Compression and streaming have been applied to two *real* plants. We have used two digitized plant models: a 20 year old *Walnut* tree (from [SRG97]) and an apple tree (from [CSKG03]). The walnut tree is 7.5m high and 5.8m wide (c.f. figure 2.7). It took two weeks to digitize using a Polhemus 3Space Fastrack electromagnetic device. We pre-processed it by fitting Bézier curves to a series of digitized points representing branches. Our representation is thus composed of approximatively 1900 branches with 6900 control points for the branches and 5800 control points

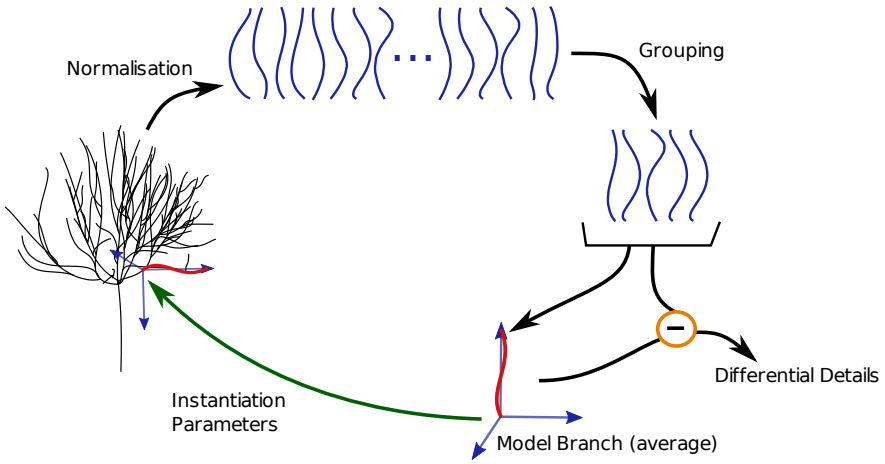

Figure 2.8 – Overview of the decorrelation process.

for the radii. The apple tree is 6 year old, 2.8m high and 2m wide and is made of 430 branches, 1350 control points for the branches and 1100 for the radii.

To extend our experimental range of models, we have also generated some examples using L-systems (c.f. [PL90]). For example, we used here a fir-like tree composed of 6 945 branches, 208 354 control points for the branches and 13 900 control points for the radii. Of course, if used in an application, L-systems models would have been surely more efficiently coded and transmitted by sending their generative rules and parameters. But determining generative process of a given tree is not always possible, in particular for measured tree.

2.3.2 Decorrelation Process

To encode a plant as a compressed multiresolution representation, we exploit the similarity of the Bézier curves representing the branches and the radii separately. The idea of the compression algorithm is to replace the absolute coding of most control points by differences compared to a small set of average Bézier curves. We group the branches and the radii independently to profit from the similarity inside each group of Bézier curves, in order to make these differences be small. Therefore they may be coded with a fewer bits, leading to a compact coding.

A simplified overview of this decorrelation process is shown in figure 2.8 for the case of Bézier curves representing branches (the process for radii is equivalent but *less visual*). First, we process a normalization transformation to make the Bézier curves comparable (section 2.3.2.1). Then we group the curves following similarity criteria (section 2.3.2.2). And for each group, we extract a *model* Bézier curve, i.e. a curve which best represents the curves of the group (section 2.3.2.3). This *model* curve allows us to express the Bézier curves of the group as two entities: *instances* and *details* (section 2.3.2.4). The *instances* depend on the normalization

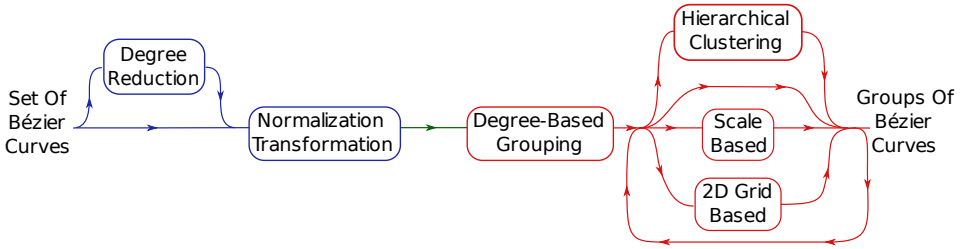

Figure 2.9 – The process of normalizing and grouping Bézier curves seen as cascading filters. All filters work on generic Bézier curves except the *normalization transformation* which has two versions, one for branches and one for radii.

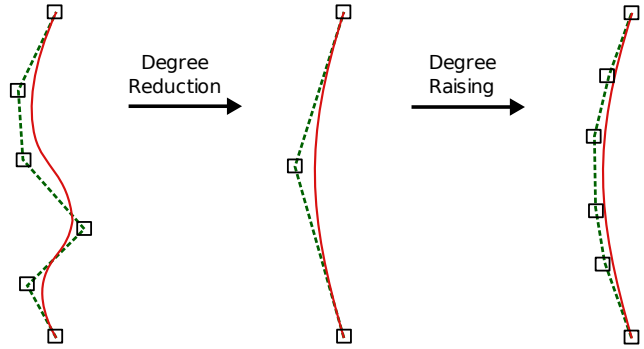

Figure 2.10 – Degree reduction and raising of a 2D Bézier curve. The original curve has degree 5 (6 points), its degree is lowered to 2 and elevated back to 5.

parameters and allow the decoder to *instantiate* the *model* bézier curves to build approximated branches and radii and place them on the tree. The *details* are the differences between the actual curves and the *model* one, details allow the decoder to *deform* the model curve and rebuild the original one.

2.3.2.1 Normalizing

In order to compare and to code differences between two branches, a so called *standard representation* of the Bézier curves is necessary. This normalization is based on two steps. The first one is optional but applicable to every Bézier curve which has a degree greater than 2. The second one, while mandatory, is different for branches and radii to profit from their particular geometric properties. Figure 2.9 (left) shows how those steps are chained as *filters*.

Optional Degree Reduction

To have Bézier curves be comparable by their control points, and release the constraint on grouping according to the degree, i.e. according to the number of

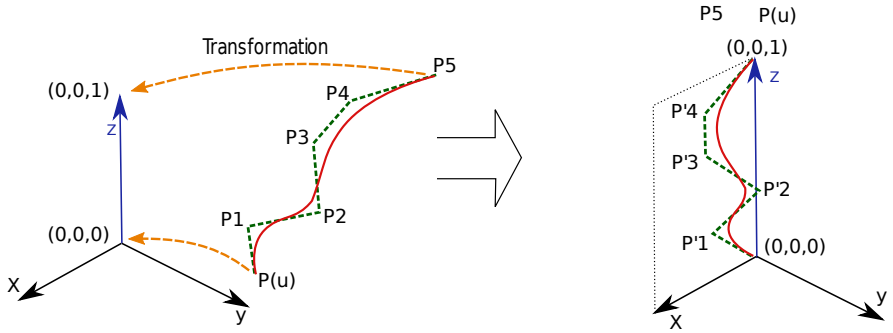

Figure 2.11 – Main idea of the normalization transformation for branches.

control points (c.f. next section 2.3.2.2 and [MCM+08, MCM+09]), we can build the standard representation by *preprocessing* the curve: we use a degree reduction algorithm (c.f. figure 2.10). In practice, any Bézier curve of degree bigger than 2 is approximated by a curve of degree 2. We apply the algorithm called "CEQ 2" from [BWX95] which is based on *Constrained Equioscillation* and has the convenient property of interpolating the endpoints of the approximated curve.

Transformation of Branches

We make all branches comparable thanks to an affine transformation converts back and forth between an original branch and its standard form. The affine transformation is defined so that the first and last control points of the original Bézier curve, map to the origin (0,0,0) and the point (0,0,1) respectively (c.f. figure 2.11).

We characterize this first mapping by a translation, two rotation angles, and a uniform scaling factor. Since we choose to apply a uniform scaling, there is a remaining degree of freedom, which corresponds to another rotation around the z axis. To completely define the affine transformation, we fix the rotation around the z axis so that the center of gravity (or average) of the control points, lies in the (x,z) half-plane. In the case of degree 2 curves, which have only one *free* point (the two other points have been fixed to (0,0,0) and (0,0,1)), this latest rotation brings all curves *totally* in the same half-plane.

Transformation of Radii

Since the parameters u_i already fall in the interval [0,1], we normalize the family r_i by dividing it by the average norm of the radii. All normalized radii profiles provide hence the same average *thickness*.

2.3.2.2 Grouping

The grouping of branches is a step in the decorrelation process which impacts the performance whole scheme. The accuracy of the approximation by *model* Bézier curves as well as the performances of the entropy coding of the *detail* vectors may depend on the quality of the grouping.

Grouping is a global function that partitions a set of normalized Bézier curves in a set of groups of normalized Bézier curves. This function can be seen as a cascade of *filters*, i.e. grouping algorithms. We have implemented several grouping filters, each to satisfy different criteria: compression efficiency, quantization error minimization, or the visual aspect of the progressive decoding. Note that these criteria apply for both the original (full resolution) plant and the intermediate, partially rendered, plants. Figure 2.9 (right) shows how we can combine those *grouping filters*. The following details the grouping schemes we have implemented.

Degree-Based Grouping

The first grouping strategy we have successfully implemented is simply based on the degree of the Bézier curves (c.f. [MCM+08]). The degree of a Bézier curve is its number of control points less one, hence, we just group the curves according to their number of control points. This approach, even if apparently *straight forward*, allows us to compare the curves without having to elevate their degree (elevating the degree of a Bézier curve adds points but does not add "information"). Moreover, as shown in figure 2.9, we still use this grouping algorithm in all cases. But we must note that, if we use the optional degree reduction step of the normalization presented in previous section (2.3.2.1), this grouping filter only partitions in two groups: the curves of degree 1 and those of degree 2. As mentioned before, the goal of the degree reduction pass, was rightly to remove the constrain of the degree on the groups.

Hierarchical Clustering

In order to minimize the quantization error (c.f. section 2.3.3.3), we add a grouping filter based on a hierarchical clustering algorithm [Joh67]. Clustering is applied on the Bézier curves by first defining an initial distance between every two curves. Then, a greedy procedure merges the clusters two by two, choosing, at each step, the smallest distance until the desired number of clusters is reached. At each merge, the distances to the newly created cluster are easily computed using a *link function*, which computes the distance to the new cluster from the distances to the two original clusters. The algorithm ends thanks to a *stop condition* based on a target number of clusters and/or a minimal *radius* of cluster.

The functions used as distance and link function have significant impact on the resulting groups. After trying several combinations, we have *converged* to the most *natural* ones given our goal. As distance we use the sum of the norms of the differences between control points; as link function we choose to compute the average distance weighted by the number of curves in the clusters. Those choices are natural, as the differences between control points will be used to compute detail vectors (c.f. section 2.3.2.4), and the model curves will be computed as an average curve of the group (c.f. section 2.3.2.3).

Scale-Based Grouping

An additional grouping strategy, called *scale-based*, uses the length of the branch or the average of the radius (both are equivalently obtained from the scaling factor of their respective normalized representation) to create the groups. We partition uniformly the segment defined by the minimal and maximal lengths (or *scales*) in a target number of segments. We then create the groups by associating the curves with segments.

This grouping filter has been specially designed for Bézier curves representing branches (even if it is usable for radii). As a typical tree has fewer long branches and more short branches, longer branches tend to be grouped in smaller groups, while shorter branches are grouped into bigger groups. Since short branches are likely not to bear children branches, having a less accurate version of these branches in the intermediate tree is visually acceptable. For long branches, the shape of a branch affects all children branches and the whole shape of the tree, they also cause popping effects. Such scale-based grouping not only preserves good compression, but gives better visual results for the progressiveness of the tree (c.f. section 2.4.1).

2D Grid-Based Grouping

Finally, another grouping strategy specialized for Bézier curves of degree 2 which represent branches has been implemented. This filter uses the geometric position of the *middle* control point of the approximating degree 2 Bézier curve. It is based on a two-dimensional grid partitioning of the (x,z) plane, taking advantage of the fact that the normalization transformation has brought the middle point in that plane (last rotation). For other degrees, or for the radii, we can use the average of the control points to process this kind of grouping. This ensures compatibility for all kinds of Bézier curves but has little less *heuristic sense*.

The Best Compromise

In section 2.4.1, we present some experiments on the grouping policies. These experiments lead us to choose a *best compromise* setup, which consists in using the degree reduction option for both branches and radii, and grouping only the branches using the scale-based filter (creating four groups). Obviously our choice can be challenged: other performance criteria or other experimental data (plant models) may produce a different best compromise. Nevertheless, we are using this *best compromise* setup for the rest of the experiments for the sake of simplicity. Moreover, changes in grouping strategy induce *small* quantitative changes, they do not perturb qualitative observations on the results.

2.3.2.3 Choosing the Model Curves

The previous process has allowed us to obtain a set of groups containing normalized representations of the Bézier curves. We can now compute the *model* curve

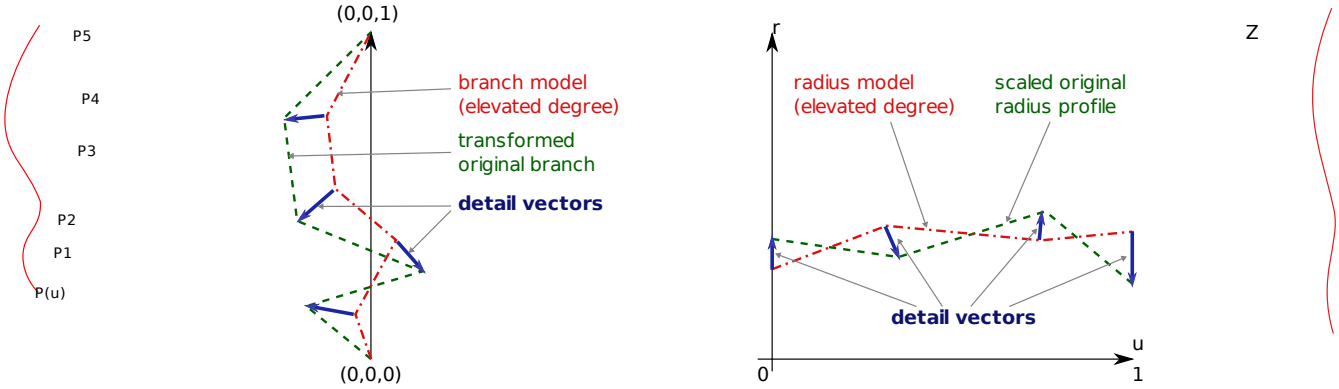

Figure 2.12 – Definition of the detail vectors for branches (left) and radii (right). In green regular dash, an *original* curve after normalization transformation, and in red irregular dash, the *model* curve of its group after *optional* degree re-elevation. The *detail vectors* (blue) are the difference vectors between the control points of the normalized original curve, and their corresponding control points on the average curve.

for each group as an average of the other curves. The average Bézier curve has the same common degree of the group (and in the case of branches, its endpoints are also (0,0,0) and (0,0,1)). Control points are computed such that the i-th control point of the average curve is the barycenter of the i-th control points of the curves of the group.

2.3.2.4 Expressing Instances and Details

For each branch or radius in a group, we now code in *differential form* the corresponding Bézier curve relatively to the model curve, storing, instead of the coordinates of the control points, their differences to the corresponding control point of the model curve (c.f. figure 2.12). We call those differences *detail vectors*. If *degree reduction* has been used for normalization, we re-elevate the degree of the model curve before computing the detail vectors. Degree raising is a deterministic algorithm (c.f. [Far02] and figure 2.10), it can be therefore used both on encoder and decoder side, and provide the same result.

For each curve, we can also define *instantiation* parameters. They are the minimal requirements which will allow the decoder to *draw* a generalized cylinder from the branch and radius models on the partially rendered tree, i.e. to place it on a parent branch cylinder. For branches we need:

- a reference to the *model* branch;
- a reference to the parent branch;
- the attachment parameter (u);
- the inverse of the (affine) normalization transformation.

For radii the case is simpler, the requirements are:

- a reference to the *model* radius;
- the scaling factor used during normalization.

All those parameters define what we call *instances* in our modeling scheme.

The encoding of a branch cylinder is now defined by five entities: the branch model, the radius model, the *instantiation* parameters, the branch details and the radius details. Next section explains how the original tree is reconstructed.

2.3.2.5 Reconstruction of the Plant

Our representation allows branches of a plant to be displayed progressively as generalized cylinders in two ways. First, the models are transformed thanks to the instantiation parameters; the resulting instances are displayed attached to their parent branch, showing an approximate cylinder of the branch. Second, the detail vectors may refine the shape of the branch previously rendered.

The approximation of a Bézier curve is built by applying the inverse of the normalization transformation to the model curve. However, algorithmic modifications may be applied to the approximated curve to improve its visual aspect. *Deterministic* modifications do not change the accuracy of the cylinder with respect to the original one just its rendering result. For example, temporary approximations of the radii profiles (while details are not available or decoded) can be made more pleasant looking just by bounding the radius for $u = 1$.

2.3.3 Binary Coding

After transforming our set of connected Bézier cylinders into a progressive representation, we obtain three *classes* of data: *models*, *instances* and *details*. We now efficiently code them to build a set of interdependent pieces of data, called *binary chunks*. Some general information, necessary for the decoder, will be agglomerated into an unclassified chunk of data: the *header*.

In this section, we first detail and classify the pieces of data we actually have to code (section 2.3.3.1). Then we focus on the "low-level" coding: first of the generic pieces of data such as numbers and vectors (section 2.3.3.2), and then of the detail vectors which deserve entropy coding (section 2.3.3.3). We finally assess the three classes of binary chunks we obtain after coding (section 2.3.3.4).

2.3.3.1 Data to code

For the main classes of data, we express here which parameters have to be coded to be able to progressively decode a plant.

The Models

First, for the branch curves, it is important to note that since the *models* are in normalized form, the first and last control points do not need to be coded. Those are always (0,0,0) and (0,0,1). For example, Bézier curves of degree 4 representing branches only need 3 intermediate points to be defined. Moreover, if we have used the *degree reduction option*, a branch model can only be of degree 1 or 2, therefore only 0 or 1 control points need to be coded (this leads to improvements in the compression ration, c.f. section 2.4).

To reference both the branch model and radius model while decoding an instance, we need to define a *model identifier*; a positive integer strictly smaller than the total number of models.

Hence, a branch model of degree d consists in $d - 1$ 3D control points and one "model identifier". Whereas a radius model of degree m consist in $m + 1$ 2D control points and one "model identifier".

The Instances

To instantiate a model branch on the progressively decoded tree, we first need to reference its parent branch, which is another *instance*. This requires coding of an instance identifier and a reference to another instance. Both identifiers are also bounded integers. Then to place the curve on its parent branch, we need the attachment parameter, which is a bounded real number ($u \in [0,1]$).

Then we need to transform the model curve of the branch. For that, as shown previously, we need first to reference the branch model of the instance by its model identifier. Then we need the normalization transformation. As seen in section 2.3.2.1, the normalization is the composition of one translation, one scaling and three rotations. However, thanks to the attachment parameter we can have the position of the first control point of the curve. We know one point, which is (0,0,0), and its corresponding translated point, given by the parent branch and the attachment parameter. Therefore we do not need to code the translation; we can obtain it from data which is already decoded. The remaining transformation parameters are three scalars for the angles of rotation and a scalar for the uniform scaling.

For the radius, we just need to code a radius model identifier and one scaling factor to transform the radius model to the right approximation.

The Details

As for the models branches, differential details for curves of degree d require the coding of $d - 1$ 3D vectors as they are differences between normalized branches. Moreover, to reference the branch to whom the details belong, we need to join an *instance identifier*.

Similarly, details vectors for Bézier curves representing radii of degree m, consist in $m + 1$ 2D vectors and one "radius model identifier".

We choose to "pack" together, in the same binary chunk, shape differences and radius differences of a given branch. Evaluation of the case when details are treated

separately is left for future work but we must note that the only *direct* overhead would be to have to code an additional instance identifier in the Radius Detail Vectors binary chunk.

2.3.3.2 Coding of Generic Data

Excluding the detail vectors, which will be studied in the next section, we only have three types of numbers to code: *general scalars*, *bounded scalars* and *bounded integers*.

- General scalars are coded with floating point representation as we do not have information on them; they can be arbitrarily big or small. Floating point is, for us, the safe "default" encoding, when we do not know enough about the number. They are the control points of the model Bézier curves and the scaling factors of the instances (one for the branch and one for the radius). For example, to remain "compatible" with strangely formed trees (with arbitrarily big ratios between small and big branches), we can not asume anything on scale factors; a scale of 0.1 is very different both from 0.0001 and from 1000.

- The attachment parameters and the rotation angles, have the particularity of being bounded and *uniformly spread* in between their bounds. Hence, as the bounding intervals of those scalars can be uniformly sampled, they can be serialized more efficiently with fixed point arithmetic. A binary integer can represent the ratio ($\in$ [0,1]) regarding the bounding interval. Therefore, we have to *choose*, for each parameter the *precision*, i.e. the number of bits used to code the number.

- Finally, bounded integers, such as identifiers and references, can be coded using a limited number of bits: *ceil(*log_2 *(MaxId))* where *MaxId* is the maximal number to code. Identifiers are regularly ordered from 0 to the number of identifiers to code, therefore this coding is optimal (i.e. the maximal number to code *is* the number of identifiers less one).

2.3.3.3 Entropy Coding of Differential Details

One advantage of multiresolution differential coding is that the induced differences are *small*, very correlated (see for instance figure 2.13). This provides the ability (i) to quantize small detail vectors with a small number of bits, and (ii) to choose accurate binary representative *symbols* according to their distribution. In this section, we first present our quantization method, then we show that the evaluation of our resulting detail vectors leads to a beneficial usage of an entropy coder.

To evaluate the accuracy of using an entropy coder in our method, we have computed, for a given quantization (i.e. a given number of bits per floating point number), the induced error and the theoretical entropy of the represented data. The

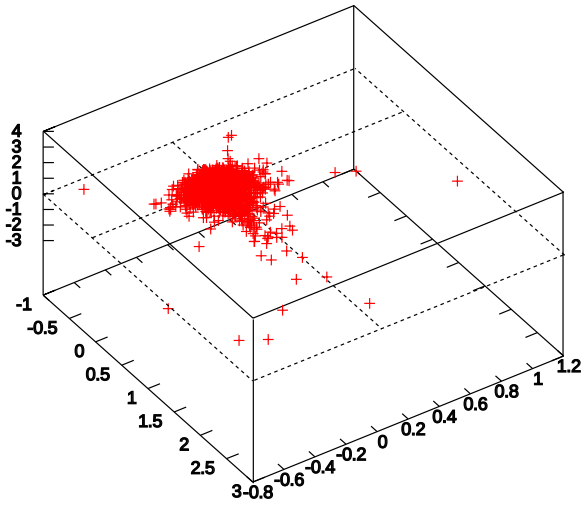

Figure 2.13 – Distribution of the detail vectors for the branches of the Walnut, using our *best compromise* set up.

maximal induced error gives the accuracy of the quantization, while the computed theoretical entropy gives the mean number of bits to expect after Huffman coding.

Quantization

The quantization can be vector or scalar. We have carried out experiments with both methods. Our first results were using vector quantization (c.f. [MCM+08]).

The vector quantization is carried out in two steps. First we compute the AABB (Axis-Aligned Bounding Box) of all detail vectors (by finding the min and max of the x,y,z coordinates). Then, to quantize each coordinate into *bpc* bits (*Bits Per Coordinate*), we build a 3D grid corresponding to $2^{3 \cdot bpc}$ vectors uniformly distributed in the AABB. Each detail vector is then represented by the symbol of the nearest of the vectors discretized on the grid. The quantization error is thus the distance from the quantized vector to the original detail vector. To reconstruct the quantized vectors, a header containing the AABB of the vectors (6 floating point numbers) and the number of bits per coordinate is sufficient.

The resulting error for a given number of bits per coordinate could still be decreased by processing a few iterations of a classification algorithm such as *k-means*. However, the resulted gain would be offset by increased header size, since transmission of the actual values of the representing symbols chosen by the classification would be necessary. We let the evaluation of the tradeof between the header size and the accuracy of the algorithm for future work, it is likely to become useful when implementing forest-based compression (c.f. 2.6), since the same header will be shared by different trees in the forest.

After analyzing vector quantization performance, we have noticed that

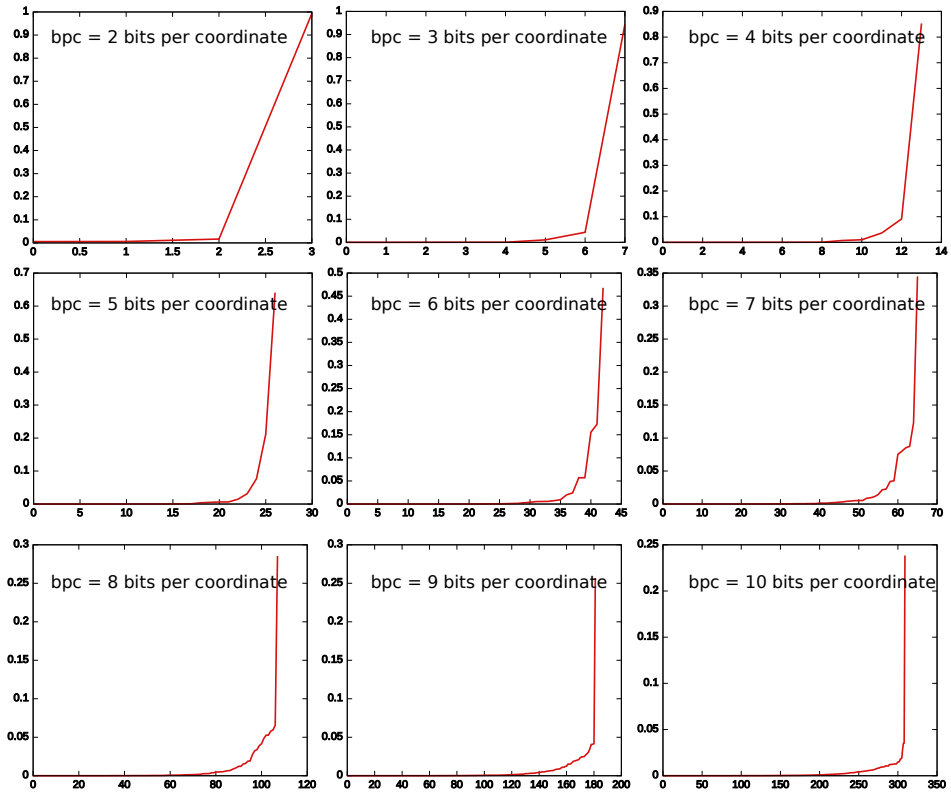

Figure 2.14 – Entropy Histograms.

- the weight of the induced entropy coding dictionary is important using vector quantization on our data; and

- detail vectors do not show a privileged direction (or a high density) and the values of their scalar components are quite close.

Therefore we have made experiments with classic scalar quantization (in other words, vector quantization in dimension 1) over the set of all scalars representing the components of detail vectors (for branches, radii and both). The results show that scalar quantization performs better. Even though vector quantization is slightly better at reducing the entropy, the gain does not compensate the higher header overhead.

Additionally, even if 3D-vector quantization was used for branches, we still would need to use 2D-vector or scalar quantization for radii. Having only one entropy coder for both branches and radii (the 1D/scalar one) allow us to code only one header and thus better absorb its overhead.

The results presented in the following section only use scalar quantization for detail vectors for both branches and radii.

bpc	**Th. Entropy**	**Exp. Entropy (w/ hdr)**	**Err. max - avg.**
2	0.094	1.012 (1.017)	0.211 - 0.059
3	0.429	1.086 (1.092)	0.105 - 0.051
4	0.768	1.233 (1.240)	0.053 - 0.017
5	1.655	1.731 (1.742)	0.026 - 0.016
6	2.455	2.469 (2.488)	0.013 - 0.008
7	3.243	3.292 (3.324)	0.007 - 0.003
8	4.037	4.082 (4.137)	0.003 - 0.001
9	4.816	4.850 (4.948)	0.002 - 0.001
10	5.695	5.715 (5.890)	0.001 - 0.000

Table 2.1 – Entropies (theoretical, experimental without and with header) and Errors (normalized against the maximal norm of detail vector: 3.995) for the *Walnut* for $bpc \in [2,10]$.

Entropy Evaluation

Once each detail vector is mapped to a symbol we can build the entropy coder. First, we build an entropy histogram, giving the number of represented scalars per symbol. We have summarized the resulting plots for one sample tree (the *Walnut*, c.f. 2.3.1.2) in Figure 2.14. To improve plot readability, we sort the symbols in increasing order of the number of detail vectors it represents, and we only show effectively used symbols. The shape of each curve shows very promising entropy coding capabilities; a few symbols represent most of the detail scalars, and most symbols are linked to zero or one scalar. We recall that the tree has been decorrelated using our *best compromise* set up.

From the built histograms we can compute the theoretical (Shannon) entropy with the formula:

$$ThEntropy = -\sum_{Proba_i>0} Proba_i \cdot \log_2 Proba_i$$

where $Proba_i$ is the normalized weight of the i^{th} represented value in the histogram (i.e. its probability).

Entropy Coding

This histogram allows us to create an Huffman coder (c.f. [Huf52]), i.e. a *tree* data-structure allowing us to assign an optimal binary representant to each vector. The information contained in the coder must be serialized and added at the beginning of the binary train, it is called the *Huffman table* and constitutes most of the *header* binary chunk.

The results obtained for *Walnut* are displayed in table 2.1 and in figure 2.15. Table 2.1 also shows the effective entropy after Huffman coding, which includes the header overhead (Huffman table and parameters). We should note that, for this

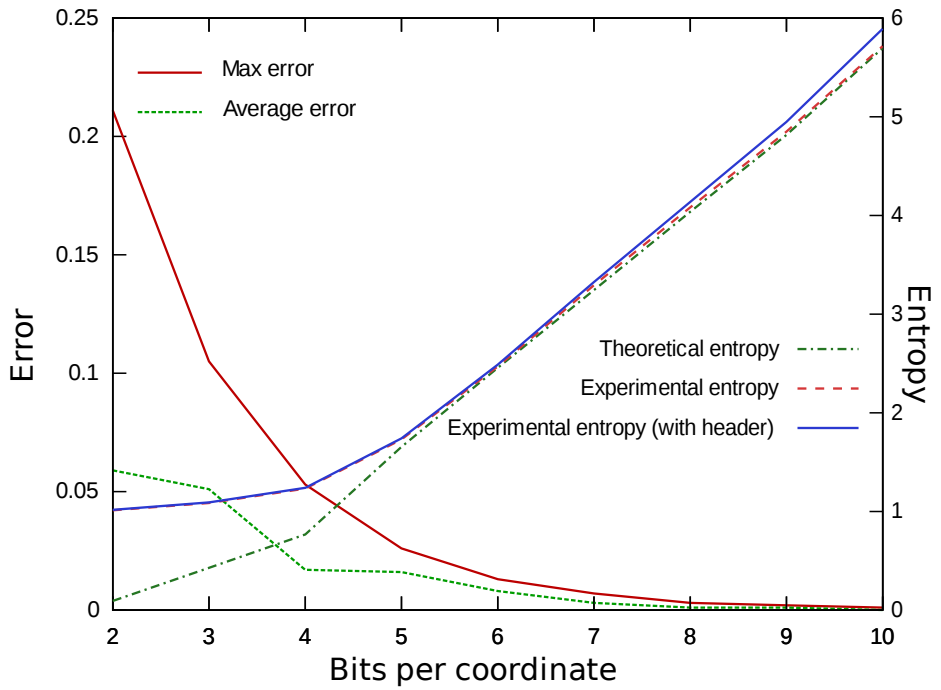

Figure 2.15 – The measured entropies and errors for the quantization of the *Walnut*.

plant, choosing $bpc = 4$ or $bpc = 5$ give a good trade off between experimental entropy and mean error, but in a *cautious and conservative fashion*, we keep $bpc = 6$ for the rest of the experiments; we prefer presenting results based on minimal error, noting that we could have slightly better compression results with a lower number of bits per coordinate. We evaluate more completely the whole compression efficiency of our method in section 2.4.2.

2.3.3.4 A Set of Interdependent Binary Chunks

The result of our binary coding process is a set of *interdependent* binary chucks. In this section, we present their format more detailedly, and then we express their dependencies.

Binary Format

There are four types of binary chunks. For k cylinders, b branch groups, and r radius groups resulting of the grouping process (section 2.3.2.2), we obtain 1 header binary chunk, $b+r$ model-chunks, k instance-chunks and k details-chunks.

For our experiments, as mentioned in 2.3.3.3, we have chosen $bpc = 6$ bits per coordinate to code the detail vectors. The number of bits to code the attachment parameter and the rotation angles (which use both fixed point arithmetic c.f. 2.3.3.2) are respectively 9 and 16. And the size of the floating point numbers is 32 bits.

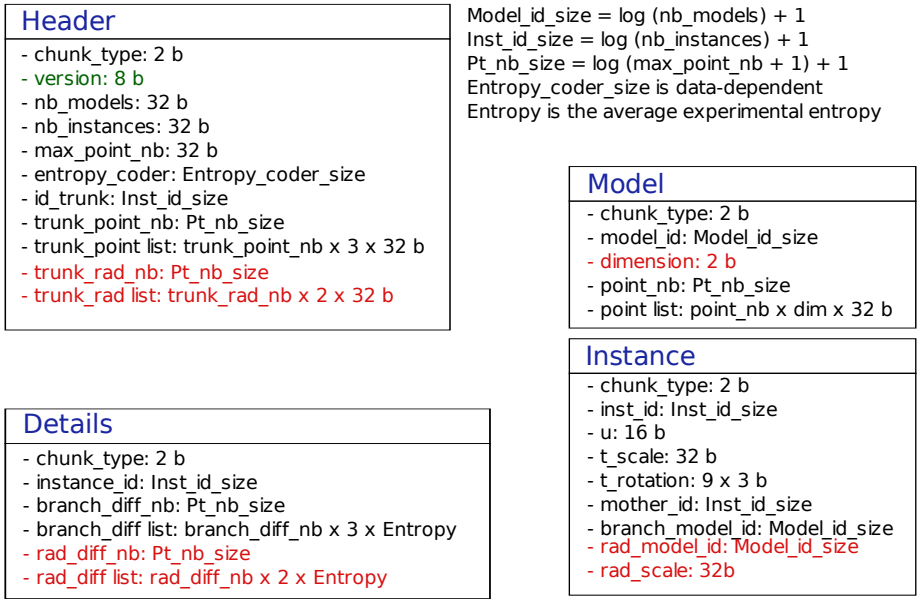

Figure 2.16 – The current format of the binary chunks

Figure 2.16 shows the exact actual format of our four types of binary chunks. To discriminate the chunks while decoding in any given order, the first two bits are used identify the type of the chunk.

The *Header* contains all the parameters needed to decode the other chunks. First, we code a version number which every binary format should contain. For now, we have two different versions (by *different* we mean "which lead to incompatible decoding"). The version 0 corresponds to behaviour presented in [MCM+08] i.e. without radii (fields written in black only); version 1 is our current format (which includes the radii, in red, also describe in [MCM+09]). Then three integers give the bounds (maxima) for the identifiers, they allow to compute the number of bits used to code model and instance references and point numbers. After an efficient serialization of the entropy coder (we code the *Huffman* tree, instead of a raw code-value table), we code the first cylinder: the trunk of the plant.

Models, instances and details are an encoding of the values presented in section 2.3.3.1. We note that the difference between radius and branch models is given by the *dimension*.

An example of result for the binary coding of the *walnut* model is presented in table 2.2 (where k, b and r are 1870, 5 and 2 respectively).

Dependencies

If we exclude the header chunk, the interdependency can be observed from the references and from the *decodability*, e.g. to decode an instance one first needs to have decoded its parent branch and its branch and radius models. There are

Type:	**Models**	**Instances**	**Details**
Number:	7	1870	1870
Min size:	12	137	27
Average size:	112.57	137.00	50.61
Max size:	204	137	251
Total size:	788	256190	94632

Table 2.2 – Binary coding results for the *Walnut* decorrelated using our *best compromise* setup (sizes are given in **bits**). The header size is 1150 bits.

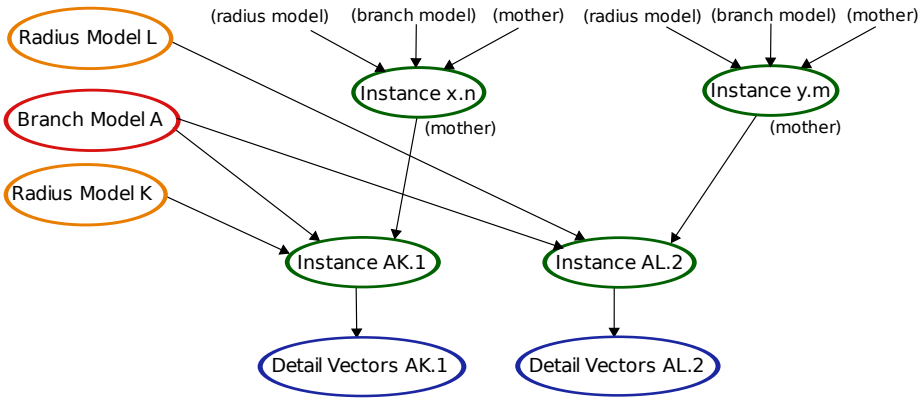

Figure 2.17 – Example of dependencies between types of binary chunks in the progressive representation.

two main families of dependencies: topological dependencies and those generated by the differential coding. The first family is related to the *n-tree* structure of the plant: a given cylinder depends on the parent branch it attaches to. The second family includes the dependencies due to differential coding, that is, on one hand the dependence between a cylinder and its branch and radius models, and on the other hand the dependence between a set of detail vectors and its corresponding instance.

The figure 2.17 is an example showing dependencies between different components of the progressive representation. Instance `AK.1` depends on its parent instance `x.n` and instance `AL.2` depends on its parent instance `y.m`; those are topological dependencies. Detail vectors `AL.2` and `AK.1` depend respectively on their corresponding instances. Note that a child instance is independent of the detail vectors of its parent instance; in section 2.3.4 we use shall this independence and show how we prioritize between children and detail vectors. Instance `AL.2` also depends on a branch model `A` and on a radius model `L`, and instance `AK.1` depends on the same branch model `A` and on a different radius model `K`.

2.3.4 Quality Metric

As our goal is to progressively decode incoming binary chunks, i.e. to provide a maximal quality of rendering for the user, we need to *schedule* our binary chunks. There are obviously no cycles in the dependencies expressed in section 2.3.3.4. Therefore a first partial order on the binary chunks derives from the dependency graph (called a *Direct Acyclic Graph* or *DAG*). This order "sends" first the chunks that are decodable, i.e. those whose dependencies have all already been sent[4].

However, this dependency-based ordering is not total. To schedule "ready-to-send" binary chunks we need a *quality metric* which will ensure that we prioritize the binary chunks that maximize the rendered quality of the partially reconstructed plant. Hence, we want to measure the visual contribution, or *importance*, of a chunk.

The visual contribution of a piece of data to a rendered model depends on the subjective perception of the user. We propose here a first tunable and easy-to-compute quality metric based on geometric considerations. We define the *importance* for each chunk as follows:

- the importance of a branch model is a constant k_0;
- the importance of an instance is the value of the scaling factor, corresponding to the *size* of the branch;
- the importance of detail vectors is the importance of the corresponding instance multiplied by the average length of the detail vectors (including branches and radii).

The next question is how to relate these three metrics to each other. The choice made is to have the importance of instances and detail vectors become both comparable with the importance of model chunks using two constants (knobs), k_1 and k_2, respectively. Intuitively, these can be chosen depending on the application and on subjective criteria. When k_1 becomes larger than k_2, the density of the tree is prioritized. When k_2 becomes larger than k_1, the accuracy of the shape of the branches is privileged. Figure 2.18 illustrates the use of these knobs, with two extreme cases.

To assess about the quality metric, we assign to the different binary chunks the following importances:

- for *models*: the constant k_0;
- for *instances*: k_1 times the instance's "size";
- for *differences*: the product of k_2, the instance's *size*, and the average length of the detail vectors.

[4]For now, we do not consider packet losses or reorderings, so, when packets are "sent" in a given order, they are decoded in that order

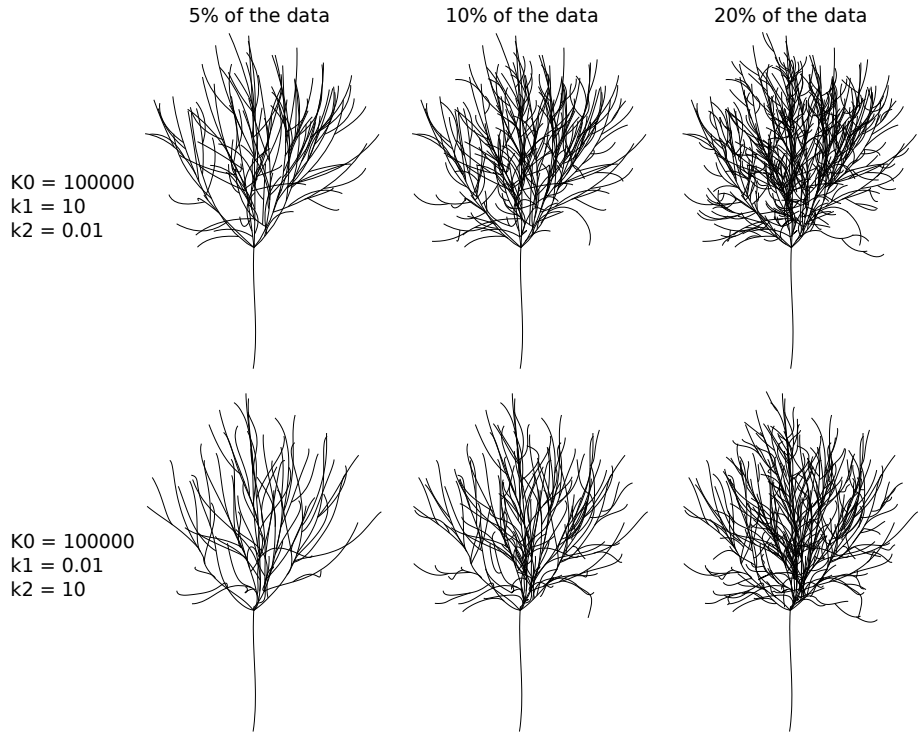

Figure 2.18 – The influence of the choice of (k_0, k_1, k_2) on the structure of *Walnut* after decoding 5%, 10% and 20% of the data.

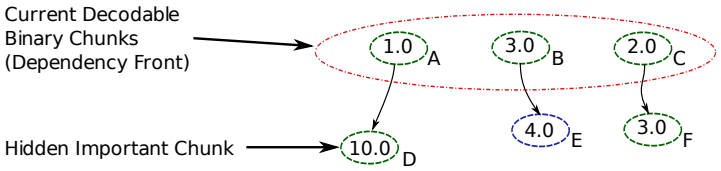

Figure 2.19 – The *Counter-Example* of non-optimality of the basic ordering policy. The ordering method (or the FIFO packetization strategy in the next chapter) will give the following order to the binary chunks: {B, E, C, F, A, D} but the total quality would have been improved faster with, for example, the order {A, D, B, E, C, F.}

The *importances* of the binary chunks, computed thanks to this quality metric, allow us to define a total order on the chunks. We define an ordering method which is *almost* optimal and provides "easy" sorting algorithm. While all chunks have not been sorted, we loop on the following steps:

- We retrieve the set of the binary chunks which are *decodable* at this point, i.e. those whose dependencies have already been sorted/sent.

- Among these chunks, we choose the one with the highest importance to be the next to be sorted.

This scheme is not *globally optimal* as there are cases where the quality can increase faster with other algorithms (c.f. figure 2.19). However, we use this method because seeking better optimality would have involved very costly algorithms, for a relatively small gain. This order will lead to the *FIFO* (First In First Out) packetization strategy in the next chapter.

Moreover, this proposed metric is for a single plant. In the context of a scene containing multiple plants, we could adjust the importance of a plant according to its distance from the viewpoint (proportionally). This importance leads to a simple view-dependent streaming: plants closer to the viewpoint may be prioritized. A more elaborate dynamic metric could also considered, for any given branches (or plant), its distance from the center of the view frustum.

2.4 Experimental Results

In this section, we show the results obtained with our progressive compression scheme. We first present experiments on the normalization and grouping setup with respect various criteria. Then we evaluate quantitatively the compression ratio induced by our method.

2.4.1 Progressive Decoding and Grouping Policies

As explained in sections 2.3.2.1, 2.3.2.2 and 2.3.4, different setups and parameters for the normalization, the grouping policy and for quality metric can be defined to optimize different criteria.

Experiments with our data made us choose a *best compromise* grouping, but one should keep in mind that some criteria may be subjective and depend on the plants on which they are applied.

For the normalization and grouping policy of the *Walnut*, we have determined three setups, for the three following criteria:

- **Best compression** is achieved by reducing the degree of branches and radii to degree 2 and not doing any further grouping. In this case the compression ratio is 3.293. This result shows that reducing the weight of the models is the best way to decrease the total size.

- **Minimal quantization error** is obtained by degree-based grouping followed by heavy hierarchical clustering. The clustering brings the most *accurate* model curves, and hence the smallest detail vectors.

- **Best visual impact of the progressive decoding**, despite being a subjective criterion, may be obtained by reducing the degree to 1 and 2, and then using

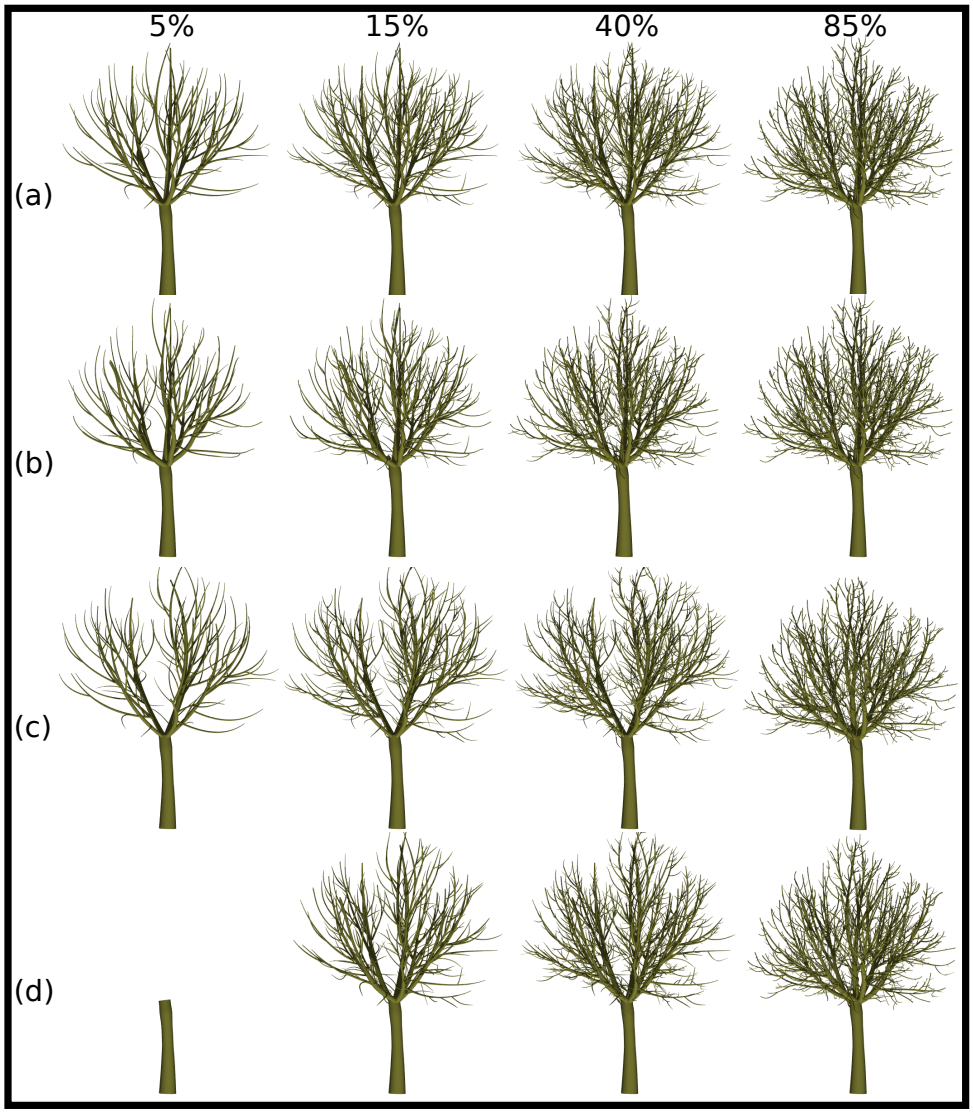

Figure 2.20 – Rendering after progressive decoding of the *Walnut*, for different setups: (a) *best compromise* with *Instances-before* quality strategy; (b) *best compromise* with *Cost-per-bits* strategy; (c) best compression with *Instances-before* strategy; (d) minimization of quantization error with *Instances-before* strategy.

scale-based grouping for the branches (and not for the radii). Degree reduction leads to very light models, therefore, the receiver has more information to decode upon receiving the same number of bits. Additionally, scale-based grouping allows bigger branches to *contribute a better approximation* of the intermediate tree earlier in the decoding process.

Tree:	Walnut	Apple Tree	L-system (fir)
Basic	143608 (1.00)	28404 (1.00)	2666968 (1.00)
Basic.bz2	84519 (1.70)	16026 (1.77)	2358353 (1.13)
PGC	44098 (3.26)	9766 (2.91)	269108 (9.91)

Table 2.3 – Comparison of coding performance of three methods: basic coding, basic coding compressed with `bzip2` and our progressive coding (PGC, *Progressive Generalized Cylinders*). The sizes are given in bytes and compression ratios (given between parenthesis) are computed against the basic coding.

As the latest setup ensures both a good compression ratio and an acceptable quantization error, we define it as our **best compromise** and use it for all the experiment results we provide (e.g. in section 2.4.2).

Regarding the quality metric, we have chosen $k_0 \gg k_2$ and $k_0 \gg k_1$, so that all models are sent before the instances and detail vectors (models are always decodable, hence their ordering, depends only on their importance). On Figure 2.20 (d), degree-based grouping is applied. Models have an arbitrary number of control points and are therefore larger than degree-two ones. A delay is noticeable: at 5% of the data, no branch instances have been decoded yet. In this case, k_0 may be lowered if visualizing models with very low percentage of the data is likely. When degree reduction is applied, the models are much lighter, and sending models first does not delay much the sending of branches (rows (a), (b) and (c)).

Moreover, for the ratio between k_1 and k_2, experiments have lead us to define two main strategies:

- The "**Instances-before**" strategy ensures that all instances are decoded before any details (priority to the *number* of branches).
- The "**Cost-per-bits**" strategy creates a relationship between the size of the binary chunks and their importance. The (k_1,k_2) knobs are chosen inversely proportional to the average size of a instance chunk and a details chunk respectively (c.f. table 2.2). This schedules the instances and details chunks so that they are interleaved given a *global sending cost*.

The figure 2.20 shows progressive renderings of the *Walnut* for the main strategies we have defined. In row (c), we notice a significant change in shape between 40% and 85% of the data. This is due to the change of shape of a major, long branch bearing many children branches. Although the grouping strategy gives a good result in term of compression for the full model, the visual quality of intermediate tree is more satisfying in rows (a) and (b) with the *best compromise* grouping.

2.4.2 Raw Compression

In order to appreciate the efficiency of our compressed model we have chosen to compare it with a well-known compression method: bzip2. For that we first con-

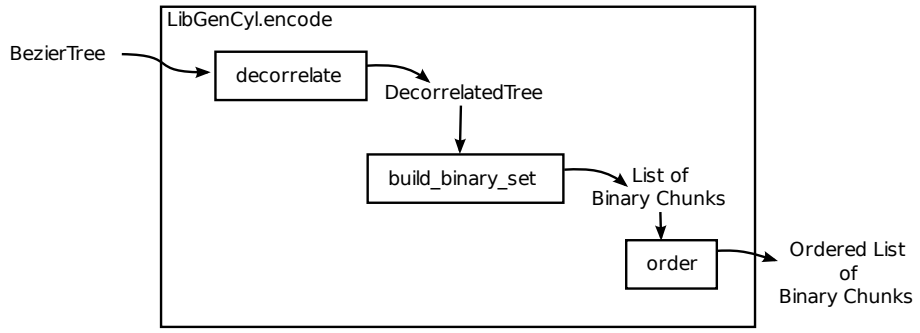

Figure 2.21 – Overview of the encoding process from a library point of view

catenate all the binary chunks to a file. We must note that if the goal was file-based compression, we could gain a little more by removing a part of the "pointer overhead": when binary chunks are concatenated, instances' and models' identifiers and details' references can be deduced from the order in the file. For instance, in the *Walnut* model we could remove at least 5145 bytes. We do not perform those optimizations as we stress on streaming of packetized plants instead of bare file compression.

Those results are shown in table 2.3. The first row contains the size of a basic serialization of geometry and topology of the connected Bézier curves (with floats and integers coded on 32 bits). The second row shows the performance after compressing the file with `bzip2`. The third row shows results for our method for the *best compromise* normalization and grouping strategy, with 6 bits per detail coordinate (*bpc* = 6 in section 2.3.3.3).

2.5 Implementation

In this section we describe the design of the library which implements the methods presented in this chapter from high-level point of view.

2.5.1 The Encoding Process

The encoding process imitates the description given in the whole section 2.3 (c.f. figure 2.21).

We start from a *BezierTree* data-structure, that is a straight-forward structure describing the generalized cylinder plant based on Bézier curves (c.f. section 2.3.1). Each branch is composed of an identifier, the identifier of the parent branch, the u attachment parameter, the list of control points of the shape and the list of control point of the radius. This structure is loaded from (and *saved to*) an XML file used as common ground to exchange generalized cylinder plants in the project. We provide export functions to several formats, for compression experiments we implement the

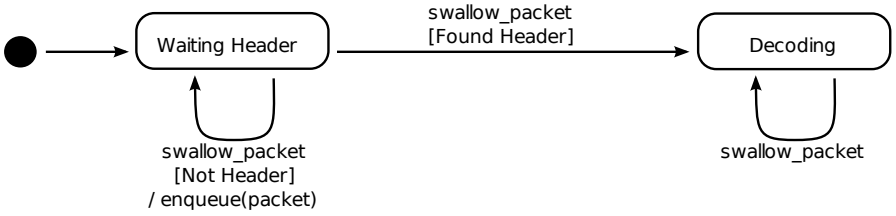

Figure 2.22 – Simple state diagram for the Progressive Decoder

basic serialization of the plant (presented in 2.4.2), for visualizing plants, we export the generalized cylinders to:

- VRML 1.0 (c.f. [BPP95]);
- the format used for generalized cylinders by the rendering platform of the NatSim project;
- SVG (Scalable Vector Graphics, [FJJ03]), a 2D vectorial lightweight rendering with and ad-hoc projection.

The first treatment is processed by the *decorrelation* module which treats separately the trunk, the branch shapes and the radii. For both kinds of Bézier curves it applies the chosen set of normalization and grouping filters as in figure 2.9. Then the *function* builds what we call a *Decorrelated Tree*. This structure contains, in addition to the trunk and some *header* information, the set of branch and radii groups, each of which contains the model curve and the corresponding instances and differences.

The build of the binary chunk set is composed of two passes. The first pass collects the information required to chose the number of bits for the identifiers, and above all, to build the entropy coder. Once the Huffman-like entropy coder is created, the second pass parses the whole structure and creates the binary chunks. In the same time explicit forward and backward dependencies are collected. Finally, the header binary chunk is written (c.f. figure 2.16). The whole set of binary chunks, is a doubly-linked DAG structure containing ready-to-send binary strings.

The last step is the ordering. The algorithm is the one described in section 2.3.4, it uses both backward and forward dependencies for performance reasons. The resulting list of binary chunks if sent in this order with packets of predefined size represents the FIFO packetization strategy.

2.5.2 Decoding Progressively

The library provides a binary chunk decoder able to decode binary packets containing an arbitrary number of binary chunks in any given order. The only requirement is that the *header* chunk must at the beginning of a packet. This is due to the fact

that the *header* is needed to be able to decode other chunks and hence to know their size.

The Progressive Decoder module consists in a two-states-machine (c.f. figure 2.22) which provides two services: “swallow packet” and “get current state”. The first function decodes incoming packets as in the figure, the second one allows to retrieve the current partially decoded tree. The format used is obviously the same as the input one.

2.5.3 Rendering Plants

We have used mainly three rendering techniques to visualize the plants.

- The first one is the SVG export described in section 2.5.1, the result can be seen (without radii for readability) for example in figure 2.18.
- The second method is a wire-frame (i.e. without radii) simple OpenGL raster. It is currently being reimplemented for mobile devices.
- The last method uses the plant manipulation platform developed at the CIRAD: OpenAlea (c.f. [PDKB+08]), and more precisely its component called PlantGL. We use OpenAlea to generate input for the POV-Ray ray-tracing engine[5] to render full images (c.f. figures 2.7 and 2.20).

2.6 Conclusion and Perspectives

We have proposed an original progressive representation of branching systems. This representation allows efficient compression of the plant geometry represented by generalized cylinders. Our method outputs a set of interdependent binary pieces of data, that are used in the next chapter for packetization and progressive transmission over lossy networks with the help of the attached quality metric.

There are several directions to extend this research. We present here some ideas for each step of the process.

The Model

Our plant models have no leaves. Our model focusses on the branching structure of the plants. To incorporate leaves, the first easy step would be to incorporate the leaves as *special branches*. We can use the instantiation scheme to encode leaf models and treat actual leaves as instances of those models.

A further option to treat the leaves is to manage independently a *global absolute distribution* of leaves on the tree. This would break the dependency that a leaf would have on its parent branch if using the previous idea. Therefore, the system would be allowed to decode and render the foliage of the tree sooner, and thus provide a more pleasant image to the user.

[5] c.f. www.povray.org

Independently of the leaves, the modeling method could be extended to support other types of curves. We have used Bézier curves because it was the model used for our input data, but most of the encoding process can be used with any kind of curve obtained from control points. For example, B-Splines or Non-Uniform Rational B-Splines (NURBS) could be used to model the shape of the branches. All the presented work can work *out-of-the-box* or with very little adaptations, except *Degree Reduction* which is optional during the normalization. In order to replace the degree reduction optimization, an alternative type of curves would need to provide two functions: (i) one function which approximates the curves with less control points, and (ii) one function which allows, deterministically, to increase the number of control points.

Normalization

On the normalization side, we would like to try alternative degree reduction algorithms, for example by enabling degree reduction to degree 4 when the branch are more accurately approximated with one *inflexion point*.

Grouping Method

We have tried several grouping methods during our study but we still could try other algorithms. We could try to group the branches following their actual shape (and not the shape of the polygon formed by their control points). For that, we could investigate the use of Principal Component Analysis (PCA), Minimal Energy Surfaces [Oss86] (to measure the *difference* between curves) or methods more specialized on curves such as Curvature Scale Space Indexing [MAK96].

To have more accurate models while keeping a good ratio between the number of models and the number of instances, we could scale the method to manage more than one tree. This forest-based scheme would profit from inter-tree grouping, i.e., curves of different trees could be grouped with more aggressive similarity criteria, and thus, induce more accurate grouping.

Progressive Reconstruction

To improve the progressive decoding of the plant and provide more branches *sooner*, we may implement *temporary inference of the models*. Thanks to lightweight parameters provided in the header, we could generate a set of *generic* curve models and use them to render the incoming instances while waiting for the models. The *real* models could, hence, be delayed to let room to instances. Those models could provide basic shapes, for example: one straight model curve of degree 2, two models of degree 3 (high and low curvature), five models of degree 4 (given the curvature and the position of the inflexion point), and so on. The overhead that may be induced by *pointer-information* should be balanced by the gain in accuracy of the coarse resolutions.

Binary Coding

At the binary coding level, two main ideas can be foreseen:

- We may first evaluate the cost of separating branch and radius detail vectors in different binary chunks. We have observed that radius details seem to carry less interesting information than branch details; radius details could be delayed in the ordering. Here again, there would be a balance between the potential gain (the accuracy of the progressive decoding) and the coding overhead induced by a different kind of chunk with more pointing information.

- To reduce the binary size of the instance binary chunks, we could employ *cheaper* differential coding for some parameters: *bit-plane coding*. For example, we might code, in the instance chunks, the attachment parameter's 5 most significant bits and, in the detail chunks, the 11 least significant bits. We would have the same final quantization error, but with intermediate *lighter* approximations.

Quality Metric

We have proposed a simple, but efficient, quality metric to assign importances to the binary chunks.

To have a more relevant choice of the k_1 and k_2 parameters, an user survey may be useful. The survey would evaluate the subjective impact of the parameters on the (animated) progressive decoding of the plants, especially in the presence of undesired effects such as as *popping*, which is a common problem of most 3D multiresolution models.

While managing forests or natural scenes, the quality metric could be made more *dynamic*. We could consider the viewpoint of the navigating user more accurately. For example, at the scene level, scene data close to the central region of the view frustum should have a higher importance

Other Applications

The efficiency of our progressive representation could be evaluated in other applications. On one hand, there other fields requiring progressive models for plants; for example 3D visualisation on mobile devices, plant modeling/sketching or animation software. On the other hand, there may be application domains that handle similar structures (connected generalized cylinders, or simply connected curves), for example respiration and sanguine circulation systems, which have plant-like topology.

Chapter 3

Packetizing and Transmitting 3D Objects

Contents

The previous study ended up with a set of interdependent binary chunks representing progressively a plant model. The binary chunks are ordered given an importance computed thanks to a quality metric. This order tries to ensure that, *if the chunks are decoded in the same order*, the quality of the progressively reconstructed plant is almost maximal during the decoding. In the case of a lossless and ordered transmission (e.g. in a TCP stream), this holds, but in the general case of a datagram-based transmission in a lossy environment (e.g. UDP), it does not.

In this chapter, we study the packetization and transmission of generic multiresolution 3D models over lossy networks, and in particular, we consider our progressive plant models and *progressive meshes*. After a brief introduction to the context of *best effort networks* (section 3.1), we rationalize the packetization problem (section 3.2). Then, we overview the previous work related to packetization and transmission of 3D content (section 3.3). In section 3.4, we present our analytical model and its application to our interdependent binary chunks. And finally, before concluding (section 3.6), we develop the experimental studies that we have performed over a *Wide Area Network* (WAN).

3.1 Multimedia, Streaming and Networks

Generally, there are two main methods for accessing content available remotely across a network. The first one is the downloading of a file, followed by its usage (visualization, computation, etc.). Download is the base of the Internet: for example, web pages are downloaded through the HTTP protocol[1] before being rendered/presented to the user, e-mails are exchanged by SMTP[2] servers by file-downloading, music and movies are massively exchanged every day through Peer-to-Peer downloading (P2P). The second method is *streaming*. Streaming multimedia consists in constantly presenting the media to an end-user while it is being delivered/transmitted. The first world-wide success of streaming, were the internet radios, for example, the *SHOUTcast* service which has been using HTTP for internet audio broadcast for more than 10 years. Now HTTP video streaming systems are also common place. Streaming media allows to access *more or less* interactively very large content, progressively; without waiting for download.

[1] HyperText Transfer Protocol c.f. [FGM+99]
[2] Simple Mail Transfer Protocol c.f. [Kle08]

In our case, as presented in the first chapter, 3D scenes are very large content and 3D walk-through is a highly interactive application. Progressive streaming is thus a natural way of accessing 3D objects in these applications.

Considering the state of today's networks, streaming media is a challenging task. Internet is based on a *best effort* network of networks, lacking any *Quality of Service* (QoS) guarantees. Internet is a packet-switched network, nodes of the network (e.g. routers) may enqueue, and hence delay, a packet at any time, and, if their queue is full, they may drop packets without any backward information to the sender. This architecture leads to the following main characteristics:

- The delay is variable (*jitter*).
- The bandwidth is variable.
- There are *random* packet losses and desequencing.

Therefore to re-ensure reliability and quality of service while dealing with these characteristics, end-to-end application-level mechanisms are needed.

With the years, thing have got obviously better; outsize pipes have been installed as well as smarter routers (see for example Random Early Detection methods [FJ93]). But problems remain, for example on wireless and/or mobile networks. Moreover, applications requirements are increasingly demanding, for example, for video streaming, applications need smaller start-up delays, smaller channel-switching delays or even multi-channel streaming.

With regard to the standard multi-layer model for networks (c.f. [Tan02]), we focus now on the transport and application levels. To handle the requirements of the networked applications (streaming-based or not), the transport layer is dominated by TCP (the *Transmission Control Protocol*, c.f. [Pos81]), and UDP (*User Datagram Protocol*, c.f. [Pos80]). TCP is mostly used for downloading and less-interactive streaming, and UDP is used for highly interactive and real-time applications.

TCP is a stream-oriented protocol with full reliability and order, and flow and congestion control mechanisms. TCP provides full-duplex (i.e. in both directions) stream communications between two connected peers. The error control is the service which ensures that pieces of data are sent and received in the same order, without losses, thanks to a retransmission scheme. The congestion and flow controls ensure that the network and respectively the receiving peer are not flooded by an amount of packets they can not handle. TCP uses a mostly AIMD congestion control scheme (Additive Increase Multiplicative Decrease) based on loss-detection: every unit of time, the sender increases its sending window by one unit, but when a loss is detected, it divides the sending window by two. Note that a packet loss is interpreted as a congestion in the network path, it is not always true, but this conservative design has proven to provide a reliable method since, at least most often, congestion *implies* packet loss.

UDP is a connectionless protocol for datagram-based communication. It provides minimal service to applications; actually, the only services are the dispatching of packets between applications (*the UDP port number*), and a spartan transmission error detection (based on a 16-bit checksum field, optionally parsed). UDP is designed for applications which know best what to do (or not) in case of packet loss, and for applications whose designers have thought that TCP's connection establishment was overwhelming compared to the data transmission needs (e.g. *Domain Name System*).

UDP may be used for example for video streaming, when the application targets internet end-users, designers have often to fall back on TCP. The problem is that UDP does not comply with subnetworks hidden by a NAT mechanism (Network Address Translation), and actually, most current "home" networks are NATs (even with often one single computer). UDP is more used when the service provider owns the whole path (e.g. mobile operator networks).

For both TCP and UDP, the Application Data Units (ADUs) need to be fragmented and/or packed together in Network Data Units (NDUs). In the TCP world NDUs are called *Segments*, in UDP's they are called *Datagrams*. The fragmentation and packing process, together with the scheduling of the packets, is generally called *packetization*. The operating system can take care of bare packetization in the case of TCP (it is actually the default behavior). But for UDP and/or applications which require fine-tunned performance, packetization needs to be tackled at the application level, i.e. while being aware of the characteristics of the transmitted data.

This is particularly the case for the streaming of 3D objects. This application has its own requirements (c.f. the first chapter), and 3D models have a special dependency structure (c.f. section 3.2.1). Next sections detail the actual packetization problem for the streaming of 3D objects.

3.2 The Packetization Problem

We consider now packet-based transmission of the binary chunks representing a progressive 3D model. Generally, since we consider binary chunks smaller than packets, we have to *pack* a given number of binary chunks (elementary piece of data from a modeling point of view, or ADU) inside one packet (transfer unit of the network, or NDU). As in section 2.3.4, these chunks can be totally ordered thanks to a quality metric. In this section, we explain more precisely the issues we tend to address.

3.2.1 Characteristics of Our Data

In this chapter, we consider that the input data is a set of interdependent binary chunks which can be the result of our progressive generalized cylinders scheme (c.f. section 2.3.3.4), or another progressive coding method for 3D data, like the

progressive meshes (presented by Hughes Hoppe in [Hop96]). For instance, most multiresolution coding of triangle meshes [AD01, AG05, Tau99, DG00b], point-based surfaces [Pau03, RL00, KB04, FACOS03] or hybrid representations [CN01] lead to interdependent pieces of data. The dependencies between their elementary pieces of data have the same "macro-shape"; they can be represented, in a generic way, by a DAG (*Direct Acyclic Graph*). Figure 3.3 shows a DAG dependency structure: if A and B are binary chunks (or packets), an edge in the graph from A to B models the fact that B depends on A, in other words, that to decode B we need to have decoded A. The main characteristic of these decoding dependencies is that coarser resolutions are important to decode finer ones.

Hence, in this whole chapter, we only require the data to be organized as interdependent binary chunks:

- chunks follow a partial order (induced by the dependencies);
- each binary chunk can be evaluated thanks to a quality metric.

3.2.2 Why Retransmitting Packets?

In the ideal case where data is received in the same order it has been sent, ordering packets by decodability (i.e. dependency) and following a quality metric (as in section 2.3.4) can be considered almost optimal. If the pieces of data (the *binary chunks*) are bound together to fill packets of predefined size by following this order, the scheme defines a first packetization strategy called **FIFO** (*First In First Out*).

This case is for example observed when the data is transmitted using a TCP stream. The *Transmission Control Protocol* ensures that pieces of data are sent and received in the same order and without losses.

But, as presented in section 3.1, generic IP networks provide only non-reliable connexion-less transmissions. Protocols which ensure lossless ordering of the data must retransmit potentially lost packets and, during retransmission of a given packet, buffer incoming packets that follow the retransmitted packet(s) in the original order. For instance, on sender side, TCP encodes the ordering of each *segment* (or *packet*) as a byte sequence number in the protocol header. On client side, if a segment is lost, the TCP stack buffers incoming segments which have a greater sequence number, until the sender detects the loss (thanks to the lack of acknowledgement after a time out) and the incriminated segment is retransmitted and effectively received.

This *buffering phenomenon* leads us to a simple observation: with a protocol like TCP which ensures lossless and ordered transmissions, when a packet loss occurs, a certain number of packets are arrived on client side but are kept by the network stack and hence not delivered to the application. In other words, some data is available on client's computer but the application can not use it to improve the visual quality of the rendered objects.

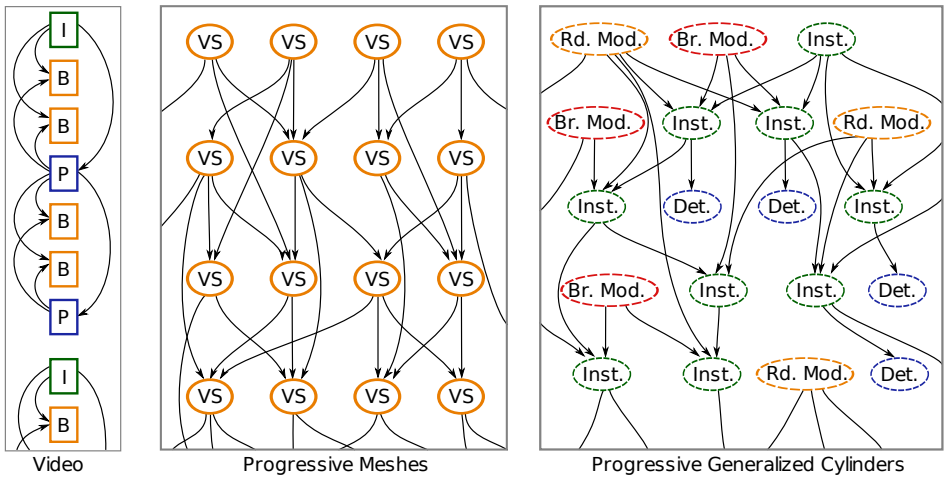

Figure 3.1 – Dependency graphs for "IBP-based" video coding (c.f. [Ric03]), for progressive meshes and for our progressive representation for plants.

For this reason, we focus our studies on datagram-based transmission protocols, i.e. without built-in ordering. We can potentially take profit from any arriving packet to improve the quality of the partially rendered model.

Commonly used datagram-based protocols bring more flexibility on the ordering but also do not hide packet losses. There are various ways to handle the packet-loss (c.f. section 3.3). We choose *retransmission*: when a loss is detected by the sender the incriminated packet is retransmitted. Losses for 3D content are hardly recoverable and may cause *durable* rendering defects. This is precisely the main difference between packet-loss for video and 3D streaming: the *persistence* (or *durability*) of the induced visual artefacts. A loss in a video stream, even for a key-frame, may have visual consequences only for a few seconds. For a 3D model, a progressive mesh or a progressive generalized cylinders plant, a *hole* in the geometry can remain visible during the whole walk-through and prevent a lot of data from being decoded. This difference is visible in the dependency graphs of these pieces of data (c.f. figure 3.1).

Our present study is thus based on datagram-based protocols with a retransmission mechanism to ensure reliability. For example, we may experiment with UDP+R (*User Datagram Protocol with Retransmission*) or DCCP+R (Datagram Congestion Control Protocol with Retransmission c.f. [KHF06]) if the help of a congestion control scheme is needed.

Those choices give a frame to our packetization study but we aim at not closing the door to FEC (*Forward Error Correction*) or to multiresolution *prediction* based on geometric, topological or numerical information (c.f. section 3.6).

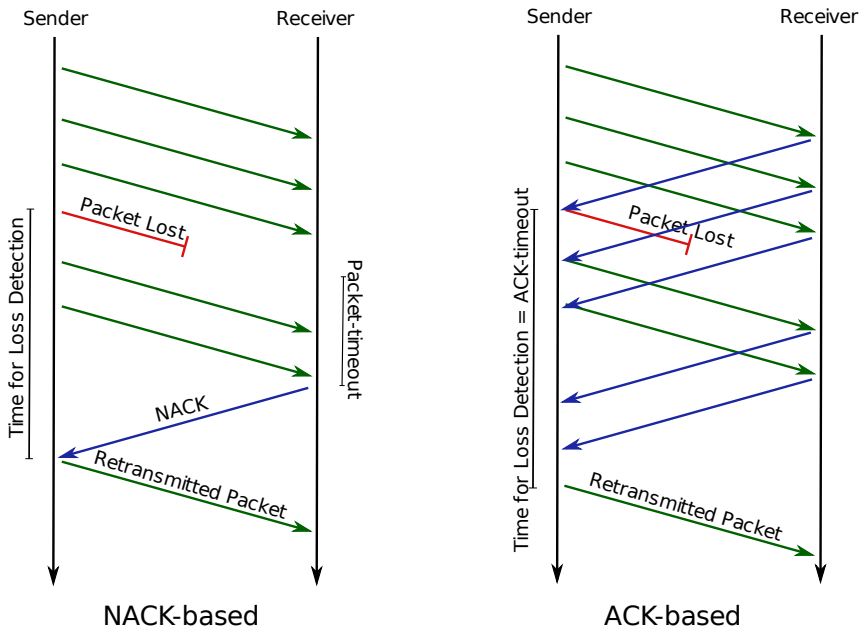

Figure 3.2 – The time for loss detection for the two main protocol cases: NACK-based and ACK-based.

3.2.3 A Dependency Problem

In this context of datagram- and retransmission-based transmissions, we focus on the order of reception of the packets. On the sender side, packets are sent in a given order but this order may be different on the client side. Two main reasons can lead to a disordering of the packets.

- Two packets may take different *routes* in the network; this case is very rare and leads generally to very small delay; it is hence not considered in our study.
- The retransmission mechanism may introduce delay for lost packets: during the time needed to detect a packet loss and to retransmit the lost packet, many following packets can reach the receiver.

There are two *main* retransmission schemes: the *TCP-like* and the *NACK-based*. In TCP, the receiver sends an acknowledgement (*ACK*) for each received segment/packet (ACK may be packed together for increasing performance). So, when a packet is sent, the sender launches a timer. If the packet is not acknowledged by the receiver before the timer expires the sender considers it has detected a loss, and schedules the incriminated packet for retransmission. On the other hand, retransmission may based on receiver's loss detection and retransmission demand. The receiver may detect the loss by a sequence number disorder and/or using also

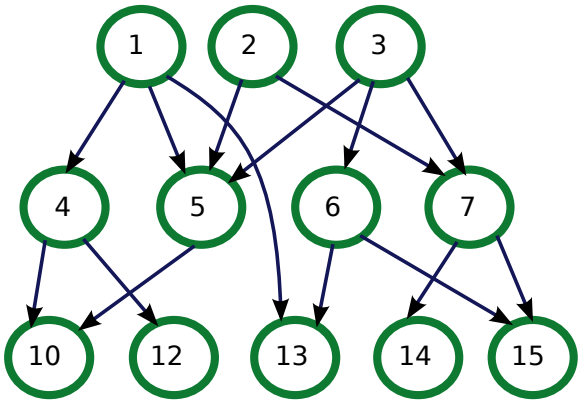

Figure 3.3 – A *generic* Direct Acyclic Graph modeling the dependencies between numbered packets. An edge from `[2]` to `[5]` (`[2]` → `[5]`), means that the packet `[5]` depends on the packet `[2]` i.e. the decoding of the packet `[5]` requires the decoding of the packet `[2]`.

a *timeout*. In this case the receiver sends a retransmission query, called *NACK* i.e. non-acknowledgement. Both retransmission methods lead to an obvious delay of the lost packet, and thus to disordering of the incoming packets on client side (c.f. figure 3.2).

The disordering of the packets induced by losses and retransmissions, becomes problematic when we consider dependencies between packets. For instance, if inter-packet dependencies are modeled like in figure 3.3, we can for example consider this (partial) sending order: `[2]`, `[3]`, `[6]`, `[7]`, `[14]`, `[15]`. If packet `[7]` is lost, packets `[14]` and `[15]` may be available on client before it has been retransmitted. This means that on the receiver side, some packets are available to the application (thanks to the datagram-based protocol) but are not decodable because their dependencies (here the packet `[7]`) have not yet been received.

Hence, the dependency problem we have to face is: **As one loss/retransmission can delay the decoding of already arrived data, we want to optimize, at any time, the amount of decodable data among arrived packets.** And thus improve the quality of the partially decoded model in a lossy environment.

As a reference to more classical problems, we may do an analogy with caching algorithms: we can define a *perfect packetization* regarding this dependency problem. The *perfect packetization* is the packetization which allows each arriving packet to be decoded immediately after reception. In other words, the one which respects the partial order induced by packet dependencies on receiver side. Obviously, as packet losses are not deterministically predictable this theoretical packetization strategy can not be expressed *before* the transmission. We need therefore a

stochastic model, in order to propose improved packetization strategies taking into account packet losses and retransmissions.

3.2.4 Dependency Vs Quality

The dependency problem previously expressed should not be considered *alone*. To maximize the quality experienced by the client/viewer, we need to take into account both the dependencies and the importance of binary chunks.

In section 3.2.2 we have presented a first packetization strategy called FIFO, this strategy attempts to maximize the quality of partially decoded model, but takes naively the dependency between packets into account. On the other hand a packetization strategy like the one proposed in [GO05] (c.f. section 3.3), which reduces the dependency between the packets, aims at addressing the dependency problem only. Since quality has not been considered such an algorithm can pack and send less important binary chunks first (which are progressive meshes' *vertex splits* in the paper), and, hence, be less efficient at increasing the quality on client side, especially when there are no losses.

Therefore, our aim in this chapter is to provide an answer to the problem of *maximizing the quality* of the decoded model *in a lossy* network environment.

Next section gives an overview of the related work before presenting our proposed scheme.

3.3 Packetizing and Transmitting 3D Objects

In this section, we give an overview of the current work on packetization and efficient transmission of 3D objects, i.e. the low-level aspect of the streaming of 3D models. Generic streaming of 3D scenes is studied in the next chapter.

Most related work focusses on the transmission of 3D mesh-based models, mainly multiresolution representations of triangle meshes. Three main classes of work exist: error resilient compression (section 3.3.1), error control (section 3.3.2), and accurate packetization (section 3.3.3).

3.3.1 Reducing Dependency

Existing work in robust mesh compression aims to reduce dependencies among the mesh, c.f. [PKL06, YKK01]. Similar to introducing key frames or restart marker in video/image coding, mesh segmentation is used to reduce the impact of a packet loss on the model. In robust mesh compression, a mesh is typically divided into several independent parts and then coded separately. Therefore the effect of one packet loss is confined within the part it belongs to. The finer the partition is, the fewer the affected vertices are. The coding efficiency, however, decreases because of more redundancies and less correlation.

As pointed out in the first chapter (section 1.1.2), we have previously proposed a 3D streaming designed for point-based 3D models (in [MMG05]). Point-based

geometry without progressive compression is inherently fault-tolerant, as there are no dependencies between surface elements (i.e. *splats*).

3.3.2 Controlling Errors

[ARAR02, ARAR05] proposed an unequal error protection method to improve the resilience of progressive 3D mesh based on CPM (compressed progressive meshes). Forward error correction (FEC) codes are added to the base mesh and additional levels-of-detail information to maximize the decoded mesh quality. The method is similar to FEC protection of video data.

Chen et al. (c.f. [CBB05]) also applied FEC to streaming progressive meshes. They analyzed several transmission schemes: TCP only, UDP only, TCP with UDP, and UDP with FEC, and studied their effects experimentally on the transmission time and decoded mesh quality. Al-Regib et al. proposed an application layer protocol, 3TP, for streaming of 3D models (c.f. [ARA03]), combining both TCP and UDP. In 3TP, important packets are sent using TCP, while the rest are sent with UDP to minimize delay. A similar combination of TCP and UDP is developed in [TYOC05] in a view-dependent scheme.

[BK02a, BK02b] proposed an "error concealment" transmission method for 3D meshes. This technique exploits the geometrical coherence of the received mesh to construct an *approximation* of the original 3D model. The model is re-sampled (pretreatment) to ensure more coherence of the data.

3.3.3 Accurately Building Packets

Packetization of various models is tackled by Harris and Karvets in [HK02]. They proposed a protocol named On-Demand Graphic Transport Protocol (OGP) for transmitting 3D models represented as a tree of bounding volumes. A key component of the protocol is to decide which bounding volumes to send. OGP begins with packing the largest possible subtree at the root and continues to pack the nodes in the subtree of acknowledged nodes in breadth-first order.

To maximize the rendered quality given the available bandwidth, [TA04] and [YLK04] propose both interleave the geometry and the texture information. [TA04] proposes a bit-allocation method which uses an original quality metric. [YLK04] uses a viewpoint-based rendering quality measurement.

Gu and Ooi (c.f. [GO05]) were the first to look at the packetization problem for progressive meshes. They model the packetization problem as a graph problem where the objective is to equally partition the graph into k partitions with minimum cut size. The problem is shown to be NP-complete and a heuristic is proposed.

3.4 An Analytical Model for Progressive 3D Streaming

Existing studies are mainly concerned with dependencies (packetization, mesh segmentation) and importance of progressive 3D data (unequal error protection, use of

Notation	Meaning
p	Packet loss rate
SendRate	Packet sending rate
DetectLoss	Time for detecting a loss on server side
$Packet_i$	The i-th packet to send
$SendT_i$	The sending time of $Packet_i$
$RecvT_i$	The receiving time of $Packet_i$
$DecT_c$	The decoding time of the chunk c
Family(c)	The dependency parents of the chunk c
$Qual_t$	The quality of the model at time t
$Importance_c$	The importance of the chunk c

Table 3.1 – Notations used in section 3.4.

reliable protocol). These two factors affect the quality of decoded models. None, however, have looked at both factors and characterize their effect on quality. We aim at achieving this by proposing an analytical model.

In this section, we present an analytical model for streaming 3D progressive data which leads to a improved packetization strategy called *Greedy*. This is joint work that has been initiated by Wei Cheng and Wei Tsang Ooi[3]. Extensive details about this model (proofs, validation experiments and simulations) can be found in [COM+07] and in [COM+09].

Section 3.4.1 presents the theoretical foundations of the analytical model. Then section 3.4.2 uses these facts to present a dependencies-and-importance-aware packetization strategy for progressive 3D models. Experiments and validation will be extensively described in next section 3.5.

3.4.1 Underlying Model

Our analytical model considers a sender sending packets at an average rate of one packet per unit time. We consider retransmission-based protocol; both NACK-based and ACK-based protocols are compatible.

Let p be defined as the packet loss rate i.e. the probability that a packet is lost during its transmission and let *SendRate* be the sending rate.

Let *DetectLoss* be the time for detecting that a given packet is lost on server-side and the time to retry to send it. Figure 3.2 shows how we define *DetectLoss* for NACK-based and ACK-based protocols. For example, *DetectLoss* for TCP's retransmission scheme is exactly the duration of the *Retransmission Timer* (c.f. RFC 2988 [PA00]).

We consider p, *SendRate* and *DetectLoss* constant. In practice this means that we use the current *computed average values*. The effects of this assumption,

[3] c.f. nemesys.comp.nus.edu.sg/projects/3dstream

which is common in network performance modeling, are evaluated precisely in [COM+07] and in [COM+09].

As *SendRate* is considered constant, we can define the *time unit* as the time to send one packet. On sender side, we define the origin (*time 0*) as the sending time of the first not lost packet. On receiver side we define the origin as the receiving time of the first not lost packet. It means that if a packet is sent at time t on sender's timeline, and not lost, it is received also at time t on receiver's timeline. And it also means that, at time t on sender side, the protocol will process the *(t+1)*-th sending.

Moreover, using this timebase on sender side and the fact that *SendRate* is constant, allow us to interpret *DetectLoss* as an integer: the number of packets transmitted between the time a packet P is sent and its loss is detected.

We note also $Packet_i$ the i-th packet to be sent ($i = 0, 1, \ldots$). The table 3.1 summarizes the notations used in this chapter.

3.4.1.1 Sending Time and Receiving Time

Let $SendT_i$ be the sending time of $Packet_i$. As $Packet_i$ can be delayed by previous retransmissions, $SendT_i$ is not i, in general. The lemma 1 shows the distribution of $SendT_i$ and its expected value given *DetectLoss* and p.

Lemma 1 *The expected sending time of packet i:*
if $i \geq DetectLoss$,

$$E[SendT_i] = (i - DetectLoss + 1)\frac{1}{1-p} + DetectLoss - 1.$$

Otherwise, if $i < DetectLoss$, *then* $SendT_i = i$.

On the receiver side timeline, let $RecvT_i$ be the $Packet_i$ is received and let

$$NbLost_{i,t} = \lfloor (t - SendT_i)/DetectLoss \rfloor,$$

be the number of times $Packet_i$ was lost when $RecvT_i = t$. Lemmas 2 and 3 give respectively the probabilities that $Packet_i$ is received at time t and that $Packet_i$ *has been* received at time t.

Lemma 2 *The probability that packet i is received at time t is:*
if $((t - SendT_i) \mod DetectLoss) = 0$,

$$Pr(RecvT_i = t) = (1-p) \cdot p^{NbLost_{i,t}}$$

otherwise,

$$Pr(RecvT_i = t) = 0.$$

Lemma 3 *The probability that packet i has been received at time t is,*

$$Pr(RecvT_i \leq t) = 1 - p^{NbLost_{i,t}+1}.$$

These results exploit the assumption that $SendT_i$ can be approximated by $E[SendT_i]$. This assumption is shown to be accurate enough in [COM+07] and in [COM+09].

3.4.1.2 Decoding Time of a Binary Chunk

In the previous section, we have expressed the sending time and the receiving time of a given packet in the protocol model. Here we introduce the dependency between binary chunks to express their *decoding* time.

We note $DecT_c$ the decoding time of the binary chunk c. And let *Family(c)* be the set containing the packet c and all the parent packets of c in the dependency graph. In other words, at time $DecT_c$ all the packets of the set *Family(c)* must have been decoded.

The probability that the binary chunk c is decodable at time t derives from the probability that one packet of the set *Family(c)* is decoded at time t and that all the other packets of *Family(c)* have been received before time t (c.f. equation 3.1).

$$Pr(DecT_c = t) = \sum_{i \in Family(c)} \frac{Pr(RecvT_i = t)}{Pr(RecvT_i < t)} \prod_{k \in Family(c)} Pr(RecvT_k < t). \tag{3.1}$$

Lemmas 2 and 3 give the expression for $Pr(RecvT_i = t)$ and $Pr(RecvT_i < t)$ respectively. Therefore we can compute an estimation of the *expected decoding time* of the binary chunk c thanks to the equation 3.2:

$$E[DecT_c] = \sum_{j=t}^{\infty} j \cdot Pr(DecT_c = j). \tag{3.2}$$

Since the probability $Pr(DecT_c = t)$ decreases exponentially as t increases, in practice we can numerically estimate the expected decoding time by considering only the first few terms of the sum. More precisely, we consider j from $SendT_i$ to $SendT_i + 3 \cdot DetectLoss$ which we found to be accurate enough for practical purposes. That is, a packet is considered to be lost at most three times in a row. For larger loss rates, one can consider more terms to trade-off computation time and accuracy.

Our analytical model is useful in several ways. These equations can help us to understand the effect of the dependencies when transmitting a progressive 3D object over a lossy network. We can also compute the expected decoded quality analytically, leading to a faster alternative to simulation as a way to evaluate the effects of network conditions on progressive 3D streaming (c.f. [COM+07] and [COM+09]).

Moreover, an improved packetization algorithm can be designed based on this analytical model. We present its design in the next section.

3.4.2 Improving the Progressive Transmission

To give an answer to the problem presented in 3.2.4, i.e. reduce inter-packet dependencies while keeping the goal of maximizing the quality of the partially decoded model, we propose in this section a packetization strategy that takes into account both the quality of the binary chunks and the dependencies between them. The main idea of the method is to give a quantitative evaluation of the trade-off between maximizing quality of the decoded model and minimizing the dependency between packets.

First, we need to define how we evaluate the quality of a partially decoded 3D model (section 3.4.2.1), then, in section 3.4.2.2, we explain precisely our proposed packetization algorithm called *Greedy*.

3.4.2.1 The Quality of a Partially Decoded Model

We consider a set of interdependent binary chunks totally ordered following their dependencies and a quality metric. In the case our progressive representation of plant models, a simple quality metric can be easily designed (c.f. section 2.3.4). If the binary chunks are the encoding of vertex split operations of a progressive mesh, various quality metrics can also be defined.

There is one constraint on the quality metric of the binary chunk: it must be monotone and independent of the decoding order. Thus, decoding binary chunks in any order always increases the total quality of the partially decoded model. In the case of our plant quality metric, we must just consider that the sum of the qualities of the binary chunks *is* the quality of the reconstructed model. For progressive meshes, metrics based on the vertex splits like the *length of the edge* may be used but, not only: metrics based on the viewpoint which give dynamic importance to vertex splits (c.f. [Hop97]) are compliant too. On the other hand global metrics like the Hausdorff distance can not be used directly in our model, since the quality is not guaranteed to be always increasing as vertex splits are decoded. Moreover, the quality would depend on the order of decoding (but we could still maybe build specialized metrics deriving from the Hausdorff distance).

Let $Importance_c$ be the importance of the binary chunk c and $Qual_t$ be the quality of the decoded model at time t. We define $IsDec_c$ such that $IsDec_c = 0$ if the chunk c is not decoded yet and $IsDec_c = 1$ if it has been decoded at time t. Then, for a total of n binary chunks, we have at time t:

$$\begin{aligned} Qual_t &= \sum_{c=0}^{n} IsDec_c \cdot Importance_c \\ E[Qual_t] &= \sum_{c=0}^{n} Importance_c \cdot E[IsDec_c] \\ &= \sum_{c=0}^{n} Importance_c \cdot Pr(DecT_c \leq t). \end{aligned} \tag{3.3}$$

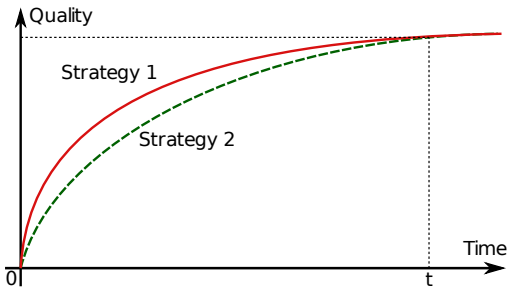

Figure 3.4 – Intermediate quality for two streaming strategies; Strategy 1 is better than strategy 2 since the area under the curve is larger.

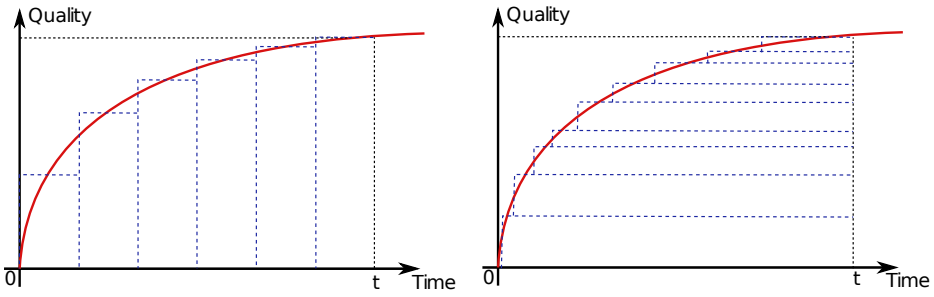

Figure 3.5 – Two interpretations of the area under the quality curve

Since $Pr(DecT_c \leq t)$ can be obtained from the equation 3.1, we are able to compute $E[Qual_t]$. Therefore, we can evaluate a given packetization strategy by predicting the expected quality curve over time given the network conditions and the properties of the 3D model.

As we want to improve the quality of the rendered model at any time, and *especially* at the beginning of the streaming session, we evaluate the intermediate quality over a period of time rather than the instantaneous one. Figure 3.4 shows that both strategies have the same instantaneous quality at time t but that still the first one is better than the second as the quality is better *sooner*. We define our evaluation metric as the area under the curve between *0* and t. As the timebase is discretized we can define it as:

$$a_t = \sum_{i=0}^{t} Qual_i.$$

As shown in figure 3.5, we can also interpret the summed quality for $i \in [0, t]$ as the sum of horizontal discrete slices. Each slice is the product of the importance of a binary chunk and the duration since it has been decoded. With this interpretation,

we can express:

$$a_t = \sum_c IsDec_c \cdot Importance_c \cdot (t - DecT_c) \tag{3.4}$$

Next section shows that the equation 3.4 allows us to define a packetization strategy thanks to a penalty criterion.

3.4.2.2 The Greedy Algorithm

Let us consider a binary chunk c. We need to decide whether we should pack c into the current packet.

First, we note that if there exists a parent of c that has not been packed, then we should not have packed c (if a parent of c arrives later than c, c cannot be decoded anyway). Thus, we only consider nodes whose parents have all been packed.

Now, consider what would happen whether we pack c into the current packet, or in the next packet. From equation 3.4, the difference in quality, $QDiff_c$, between the two cases is:

$$QDiff_c = Importance_c \cdot (E[DecT_c^{next}] - E[DecT_c^{current}]). \tag{3.5}$$

The metric $QDiff_c$, i.e. the *penalty*, is computed thanks to equation 3.2, which gives the expected decoding time for the two cases. Minimizing the penalty maximizes the difference in decoded mesh quality (equation 3.4).

Equation 3.5 captures the trade-off between the importance of the binary chunk c and its dependencies. The penalty increases as $Importance_c$ increases. Considering the case where c has a parent in the current packet, if we pack c in the next packet, then the expected decoding time for c increases; not only because it will arrive later, but also because this packing introduces a dependency between the current and the next packet. From the equation 3.2, we can see that additional dependencies increase the decoding time since both packets have to be received before c can be decoded. Therefore, increasing dependencies increases the penalty.

We can now describe a greedy algorithm to packetize progressive 3D content, represented as interdependent binary chunks. The algorithm simply packs the node with highest penalty at each step. Technically this is a heuristic since it does not guarantee an optimal packetization. The packetization problem has been shown to be NP-complete [GO05].

The figure 3.6 shows the pseudo-code for building one packet using our algorithm. The algorithm is somewhat similar to the FIFO one presented in 2.3.4: we maintain a set of *decodable* nodes among which we choose the best candidate for packetization. The best candidate is selected thanks to equation 3.5 (with FIFO we use only the importance instead of $QDiff_c$).

To reduce computation cost, we approximate $QDiff_c$ by computing $E[DecT_c]$ (for both cases) up to a limited number of terms. We choose to use up to $3 \cdot DetectLoss$ terms in our implementation. Further, from our observation (c.f. [COM+07] and [COM+09]), since the effect of dependencies diminishes with time, we stop

```
function BuildNextPacket is
    H is a maximum heap
    for all c : chunks whose parents are already packed do
        compute penalty QDiff_c
        insert c into H with QDiff_c as key
    end for
    while H is not empty and packet is not full do
        pop j from H
        pack j into packet
        for all k : children of j whose parents are already packed do
            compute penalty QDiff_k
            insert k into H with QDiff_k as key
        end for
    end while
    return packet
end function
```

Figure 3.6 – The **Greedy** algorithm.

running the algorithm after time $3 \cdot DetectLoss$ and simply send the vertex splits in decreasing order of their importance (FIFO order).

3.5 Performance Evaluation

In this section, we describe the experimentations we have performed on the transmission of progressive 3D models.

3.5.1 Goals and Problems

3.5.1.1 Objectives

Our aim is to provide experimental results on the transmission of progressive 3D models (section 3.5.1.2) over a *real-world* network (i.e. internet) (section 3.5.1.3). We have performed experiments with different goals:

1. Validate the proofs and the assumptions of analytical model.
2. Compare the efficiency of the Greedy and FIFO packetization strategies, with both progressive meshes and progressive plant models.
3. Experiment the transmission with the Greedy algorithm with constant rate and congestion controlled transmissions.

We present here the results for the last two objectives concerning the Greedy algorithm. The validation of analytical model can be found in [COM+07] and in [COM+09].

3.5.1.2 Progressive 3D Data

We have used two progressive modeling schemes: Progressive Meshes (c.f. [Hop96]) and our Progressive Plants Models (c.f. section 2.3). Both schemes pro-

vide a base model (the *base mesh* and the *header chunk* respectively), and a set of interdependent binary chunks. We also equip the progressive models with a quality metric to order the binary chunks given their importance. For the plant models, we use the quality metric designed in section 2.3.4. For the vertex splits constituting the binary chunks of a Progressive Mesh, we use the *edge length* metric (i.e. the importance of a vertex split in the progressive mesh scheme, is the length of the collapsed edge).

3.5.1.3 Collecting Traces

On the networking part, we have processed our experiments on both LAN and WAN IP-networks. We have experimented 3D streaming between Toulouse, France and Singapore. For reproducibility, we have recorded packet traces of lengthy transmissions between the *sender* and the *receiver*, and then we *play* these transmission traces off-line (measurements, decoding of the models, *etc.*). We use both UDP and DCCP with a custom retransmission scheme, we call these reliable protocols UDP+R and DCCP+R respectively. UDP packets are sent at a constant rate as required by our analytical model. The retransmission schemes are, for these experiments, implemented off-line too, i.e. the network traces are bare UDP or DCCP and then while *playing* the transmission we process the loss-detection and the retransmission. We can therefore use exactly the same network trace for different experimental setups (different models, packetization strategies, retransmission schemes, *etc.*)

3.5.1.4 DCCP Over The World

The Datagram Congestion Control Protocol is a *recent* transport protocol described in the RFC 4340 [KHF06]. With respect to functionalities, DCCP lies somewhere in between TCP and UDP. It provides bidirectional unicast connections of congestion-controlled unreliable datagrams. DCCP has been designed for applications that transfer fairly large amounts of data, but can benefit from or simply accept packet-losses and/or reorderings. DCCP aims at being TCP-friendly, i.e. its congestion control mechanism is supposed to ensure that the DCCP connection is a *good citizen* in the network and does not *steal* bandwidth from TCP streams.

TCP ensures that the output bandwidth is *compatible* with the network conditions: regarding the congestion of the network (measured thanks to loss-detection), TCP adapts its throughput following an "AIMD" (Additive Increase Multiplicative Decrease) algorithm. A *bad citizen*, like UDP could be, uses a high throughput regardless of the congestions. Hence, *a bad citizen* provokes even more congestion which makes concurrent TCP streams decrease their throughput. That is called "being TCP-*unfriendly*".

Therefore, with DCCP+R we have a protocol similar to UDP+R but with additional built-in connection management and, above all, a congestion-control mechanism. The connection management handles, for example, the connection

hand-shake, the full-duplex scheme. The congestion-control method adds the TCP-friendliness.

Two main congestion control mechanisms have been implemented in DCCP, they are referenced as CCID2 and CCID3 (*Congestion Control Identifier*). CCID2 implements TCP-like congestion control (c.f. RFC 4341 [FK06]) whereas CCID3 uses TFRC (*TCP-Friendly Rate Control*); defined in RFC 4342 [FKP06]. Both congestion control algorithms have different *notions of fairness*, the first one means "act like TCP", the second one just "follows the statistical behavior of TCP". However, we have only been able to use DCCP with CCID2 for our experiments; we have observed that the CCID3 implementation is still not reliable (at least as implemented in the Linux kernel 2.6.24).

In addition to TFRC implementation problems, we have experienced more problems with DCCP routing: there are intermediate routers (in the backbone network) between Toulouse and Singapore which do not route DCCP traffic. More precisely we have observed that IP packets tagged with DCCP's protocol number (which is 33) are dropped in their way from Singapore to Toulouse. Therefore, even the connection *hand-shake* of the protocol can not succeed. Next section shows how we get around this problem.

We were disappointed (but maybe not *surprised* ;-)) to find out that even if both academia and industry advocate the use of TCP-friendly transport protocols, DCCP, which is the only standardized one, is still by far a second class citizen in today's internet. *Internet Service Providers* do not really provide an Internet Protocol service.

3.5.2 Tools to Handle Problems

In this section, we present a tool we have developed. It is called *OMAN* (for *OMAN Measures Any Network*) and it is designed to record network traces for UDP, TCP, and DCCP protocols. Additionally, since we experienced problems with UDP firewall traversal and DCCP over the WAN, the tool provides an UDP-tunneling service to *hide* any transmission (connected or not) in a sequence of UDP packets, which preserves network conditions visible to the encapsulated protocol (loss-rate, *RTT*, *etc.*).

3.5.2.1 An UDP Tunnel

Problems with DCCP introduced in section 3.5.1, led us to implement a full-duplex UDP tunnel that encapsulates DCCP packets. Figure 3.7 shows the tunneling mechanism for one half of the full-duplex connection, taking the example of DCCP.

At the sender, our tunnel captures all outgoing packets thanks to `libpcap`[4]. Then it parses the beginning of the packet headers to discriminate the desired connection using the protocol identifier and, if needed, the port numbers. The tunnel

[4] c.f. http://www.tcpdump.org/

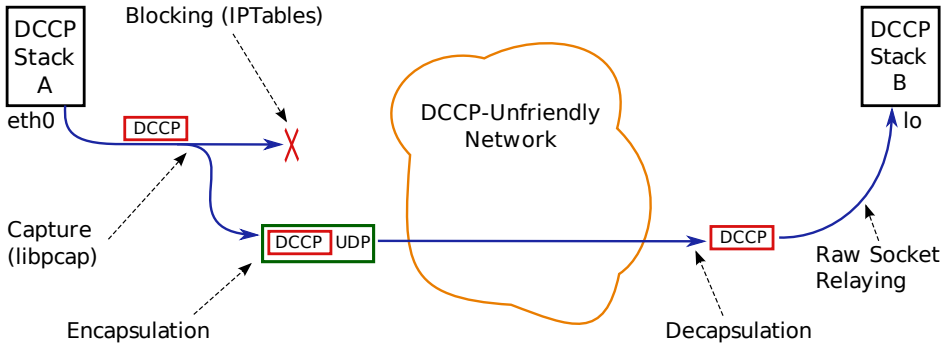

Figure 3.7 – One half of the UDP tunnel for a DCCP connection.

copies and encapsulates the packets resulting of the previous *filter* in UDP packets, and transmits them to the receiver.

At the receiver, the tunnel *decapsulates* the UDP packet and sends the original packet through raw socket to the actual receiver on the local network interface (i.e. *localhost*). Hence, the protocol stack of the receiver's operating system treats the incoming packet as if it would have been sent directly.

We have also to avoid duplicated packets that occur because of the copying of packets. Therefore, we block totally the actual non-tunnel connection by fire-walling (c.f. figure 3.7). We use Linux's built-in firewall: *Netfilter* (a.k.a. `iptables`).

We must note that capturing network interfaces' traffic, as well as sending through a *raw socket*, require `root` (i.e. *administrator*) privileges on both receiver and sender hosts.

OMAN's tunnel is able to transport UDP, DCCP and TCP (as filtering packets requires to parse header, the tunnel can not be totally "transport protocol agnostic"). Hence it can be applied to different experiments. For instance, it allows us to test a TCP transmission with "UDP network conditions". We observe that routers often privilege TCP segments over UDP datagrams in their waiting queues. That experiment still adds constraints: one needs to set a smaller "maximum segment size" to ensure that TCP segments fit inside UDP datagram payloads and to take care of the *Large Segment Offload* (LSO) mechanisms. LSO may be implemented (and activated) by modern TCP stacks and network cards and prevent the tunnel from working[5].

Moreover, in OMAN, we also use our tunnel to *spy* the DCCP protocol by decoding packet headers. We can therefore for example use DCCP's acknowledgement mechanism to compute the RTT as *seen* by the DCCP stack for the operat-

[5]TCP's LSO is a common *hackish* technique for lightening TCP management in the Operating System by sending large segments to the network card which will make them fit in the network's transfer units. It is more precisely described in Wikipedia's entry "Large_segment_offload" and in [FHL+05].

ing system (acknowledgements are, otherwise, not visible from user-level applications).

3.5.2.2 Traffic Generation

Generating traffic on server side consists simply in sending a long list of dummy packets. For TCP and DCCP the sending rate is imposed by the congestion control mechanism of the protocol, but for UDP we need to set the sending rate *by hand*. We send *nb* packets of size *psize* bytes every *period* milliseconds, to set the sending rate to *nb·psize/period KB/s*. We may call this method *burst-based* sending scheme. It allows us to increase the bandwidth while keeping *period* greater than *10 ms*. This condition ensures that timers are *reliable* and *accurate* even on a reasonably loaded GNU/Linux computer.

The client can optionally send acknowledgements in the case of the UDP and DCCP protocols. We do need acknowledgements, even if they are useless for the generated transmission, to simulate realistically UDP+R and DCCP+R dialogs. For example, while *replaying* a given trace we must consider that a packet is lost *in practice* on server side in three different cases:

- the packet is actually lost during its transmission;
- the acknowledgement has been lost during its transmission;
- none are lost but this particular round-trip time (instantaneous RTT) has been too long; server side timers have expired.

Those mechanisms allow us to record realistic network traces of arbitrarily large data sets, for ACK-based retransmission schemes, and for NACK-based ones (which are easily approximated since they are a subset of an ACK-based dialog).

3.5.3 Results and Observations

In this section we present some experiments we have processed, related to the streaming of 3D models using our Greedy packetization strategy. For particular simulations and experiments related to the validation of the assumptions of our underlying model refer to our papers [COM+07] and especially to [COM+09].

The rest of this section presents the experimental setup and then extracts highlights of the results we have obtained on WAN-based experiments. But before, we give general remarks about the network conditions we have observed.

3.5.3.1 General Observations

We have processed extensive experiments and measurements between Toulouse and Singapore during the summer 2008. From these results we can do some high-level observations.

Our first observation is that in the case of UDP+R, when the sending rate increases, the loss rate increases as expected but losses are *bursty*. The loss rate fluctuates over time in an unpredictable way. Moreover, loss detection depends a lot on the timer scheme used and on its setup. Many packets may be seen as lost by the protocol implementation but are actually just very *late*. The RTT can vary dramatically fast.

On the other hand DCCP+R experiences very few losses thanks to its TCP-like congestion control scheme (CCID2). However, as stated in section 3.5.1, we did not manage to experiment successfully with TFRC implementation. This congestion control scheme, during the *best* cases, only led to a decreasing sending rate until the transmission stalls.

During august 2008, a huge event with worldwide consequences has modified dramatically the network conditions between Toulouse and Singapore: the *Games of the XXIXth Olympiad*. The Olympic Games took place in Beijing, China, from August 8 (except football, which started on August 6) to August 24, 2008. During this period, internet users have generated a huge increase of the network traffic; a great part of the 302 sporting events were made available through video streaming. On our side, we have observed a 14% loss rate, with very regular packet losses. By inspecting the phenomenon with ICMP requests (i.e. `traceroute` and `ping`), we found that the latest routing node before the link between Australia and the United States of America was generating these losses. The surprising regularity of the packet losses may be explained by different hypotheses which we can not verify. For example one can explain regular packet losses as a *voluntary packet dropping* scheme implemented by the incriminated router; its strategy would be to provoke packet losses in UDP streams to preserve room for TCP streams (which may be considered to have a higher priority). On the other hand, as in such important internet nodes, our generated traffic is interleaved in a huge amount of packets, we may *see* loss-bursts as individual packet losses.

Finally, in addition to the DCCP routing problems we have observed (c.f. section 3.5.1), we note that *small* UDP packets (with a payload smaller than 10 or 8 bytes) may be dropped certain routes even if these routes are not congested. As a consequence, our acknowledgement messages suffered from unrealistic loss rates. Therefore, we have performed our experiments with bigger acknowledgement packets by padding the payload with about 40 bytes of useless data.

3.5.3.2 Experimental Setup

Once the tools are developed, our experimental setup is quite simple:

- collect traces of packet-based transmissions;
- run the experiments on the collected traces;
- process the measurements.

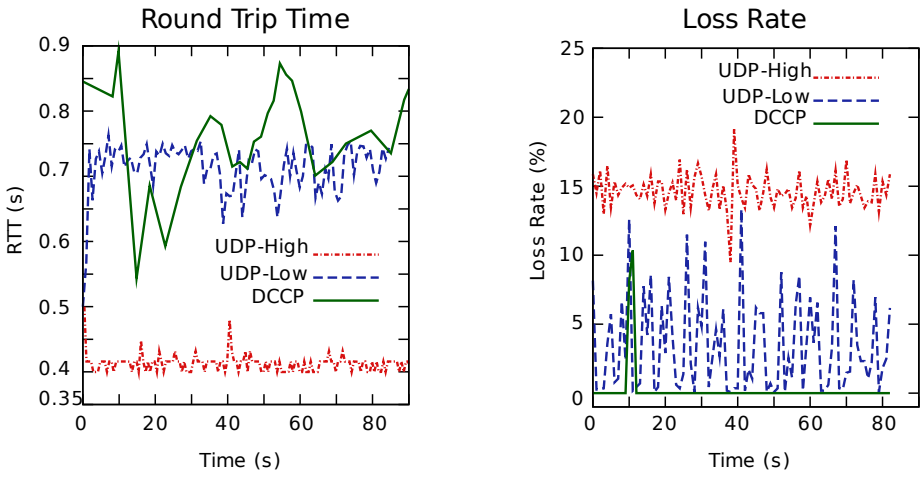

Figure 3.8 – Loss rate and RTT of the selected network traces.

For example, if we want to compare the two packetization strategies: FIFO and Greedy with respect to the quality of the rendered (i.e. decoded) model on client side. Both strategies are *functions* which take as argument a set of connected binary chunks with importances, and which return a sequence of packets to send. We *run* the network traces on the sequences of packets while measuring the quality of the decoded model at each time slot. Both strategies have hence been tested on exactly the same packet losses and the same delays. We can then plot the quality curve for both strategies.

Note that from one collected network trace, we can run a lot of different experiments. With a different *start point* (i.e. first packet sent), we get naturally a totally different loss/retransmission scenario. In the next section we use this property to run thousands of experiments from the same (long) network trace.

3.5.3.3 Validation of the Greedy Strategy

We present now some results obtained during the summer 2008 for the paper [COM+09]. These results allowed us to experimentally validate the performance of the greedy strategy.

Models and Traces

We have used several progressive mesh models, we provide here results for the *Happy Buddha* from the Stanford 3D scanning repository. It is composed of 543,652 vertices and 1,631,574 faces; once transformed into a progressive mesh, the base model has 4,703 vertices, 9,818 faces, and weights 174.268 KB, the vertex splits weight almost 20 MB.

From the large set of collected traces we chose three main traces for these experiments:

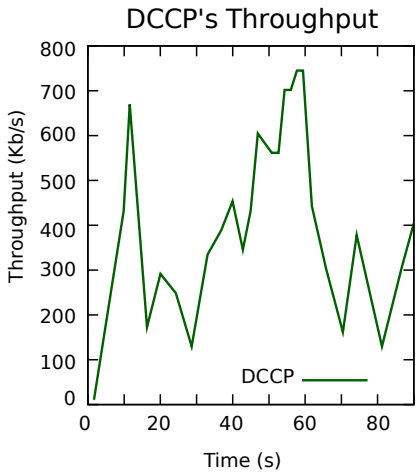

Figure 3.9 – Throughput of the DCCP collected trace.

Trace	UDP-Low	UDP-High	DCCP
Length (s)	537.1	85.2	460.3
Loss Rate (%)	14.53	3.6	0.085
Sending Rate (kb/s)	2,087	13,204	488
RTT (ms)	412	717	660
DetectLoss	77	845	29

Table 3.2 – Main characteristics of the transmission traces used in section 3.5.3.3.

- `UDP-Low` is a trace of an UDP transmission with low loss rate;
- `UDP-High` is a trace of UDP transmission with high loss rate (average: 14%), observed during the 2008 Olympic Games (c.f. section 3.5.3.1);
- `DCCP` is a DCCP trace using TCP-like congestion-controlled (CCID2) throughput (c.f. figure 3.9).

We plot the loss rate and RTT of a 90-second segment from each of the three traces in figure 3.8. The main (average) characteristics of the traces are shown in table 3.2.

Results

We now compare the relative improvement of the proposed greedy algorithm with respect to the FIFO algorithm on the *Happy Buddha* mesh with the previously presented network traces. We run one thousand different transmissions for each packetization strategy.

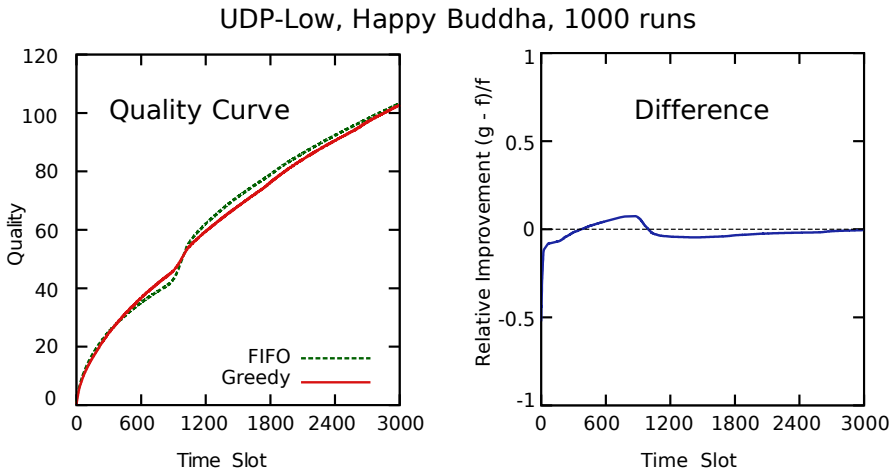

Figure 3.10 – Quality curve for 1000 transmissions of the Happy Buddha model using FIFO and Greedy on the UDP-Low trace.

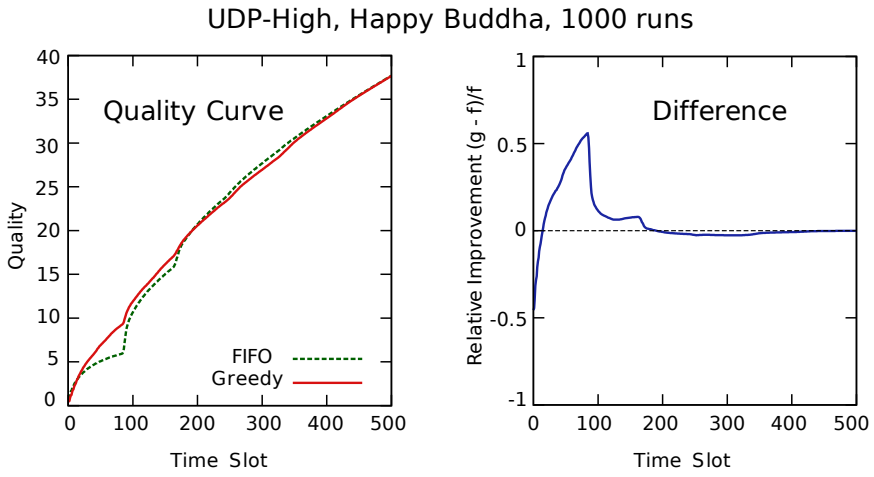

Figure 3.11 – Quality curve for 1000 transmissions of the Happy Buddha model using FIFO and Greedy on the UDP-High trace.

We define the relative improvement as *(g-f)/f*, where g is the quality of the mesh if transmitted using the greedy method, and f is the quality of the mesh if transmitted using the FIFO method.

The results, shown in figures 3.10 and 3.11, show that our greedy method outperforms the FIFO in the first several round trip times, on the `UDP-High` trace. They also show that when packet loss is low (the `UDP-Low` trace), there is no real gain to try to improve over the FIFO method. The results for `DCCP` trace are not plotted because this trace has negligible packet losses, and thus, the two algorithms work equally well (superimposed curves, zero difference).

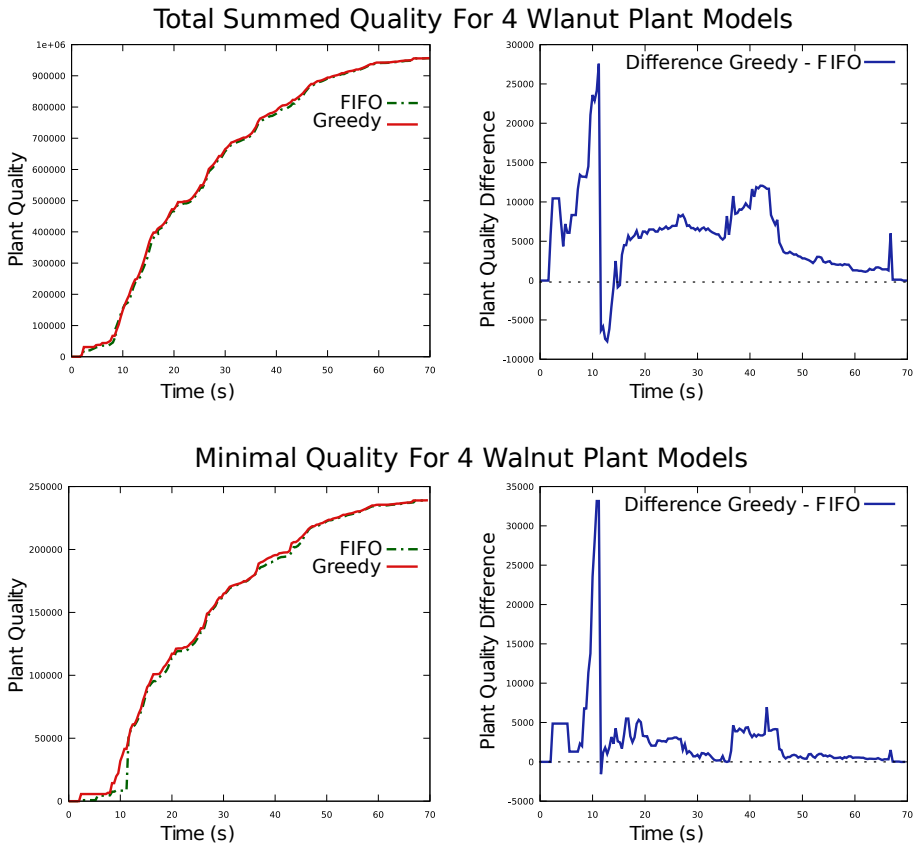

Figure 3.12 – Quality curve for one transmission of four *Walnut tree* models using FIFO and Greedy on a DCCP transmission.

3.5.3.4 Example With Plant Models

The analytical model had been initially designed for the transmission of progressive meshes. Here we show the accuracy of the analytical model on other multiresolution 3D modeling schemes, and, at the same time, at proving the appropriateness of the plant compression method presented in the previous chapter to efficient streaming. Therefore we have processed experiments for the paper [MCM+08] transmitting a set of trees over a WAN using DCCP and UDP+R.

On figure 3.12, we show an example of results obtained for the transmission of four walnuts between Toulouse and Singapore. The trace comes from a DCCP capture (with CCID2) with a 12% packet loss (extracted from a very irregular capture with that high loss rate during a 2 minutes burst). We plot total quality (for the 4 trees) and the minimal quality (at time t the quality of the poorest tree in the group).

Figure 3.13 shows the reconstruction of one of the trees after receiving 7% of its data during both transmissions. We can observe that most of the data received in

Figure 3.13 – Example showing the structure of the same tree of the scene better packetized by the Greedy strategy.

the FIFO case is unusable due to the lack of one or more binary chunks on which many others depend. The aim of the Greedy packetization strategy is rightly to prevent these "accidents" to happen.

3.6 Conclusion and Perspectives

In this chapter, we have shown how we can transmit interdependent pieces of data resulting from progressive compression of 3D models over *best effort* lossy networks. The scheme presented in the previous chapter is one example of progressive model we can packetize using the analytical model we have presented.

Experiments show that the presented Greedy packetization strategy outperforms the FIFO one, when the loss rate is high and during the first few RTTs. On interactive walk-through applications, these first seconds are important in the sense that a user, while navigating, will pass near many 3D objects, that will be visible during short periods of time.

On this topic, further work could focus on various aspects.

Reactive Management of the Retransmission Waiting Queue

Even if its presence is not *directly* compatible with our analytical model (because of constant rate and RTT), in practice, the sender has to manage a retransmission queue. In other words, generally, on sender-side, there is a First In First Out data-structure for the packets which have been detected as lost.

It could be interesting in this context to evaluate which packets (or even which binary chunks) should be retransmitted first, given the dependency, the quality *and* the current state on client side. For example, at a given instant, retransmission of a

less important packet, could *"deliver"* important packets which are not decodable on receiver-side because of missed dependencies.

Geometric and Topological Prediction

The dependency problem may also be solved or limited at the *modeling level*. One could try to predict, on receiver-side, generic replacements for missed dependencies. For example, with progressive meshes, we may try to infer lost vertex splits. While waiting for their retransmission, we would be able to render depending vertex splits as a coarse temporary approximation.

FEC Integration

Following the same idea, Forward Error Correction (FEC) could help at temporary (or not) dependency handling by adding additional data. The additional problem that we might have to deal with, is that the overhead of FEC has negative impacts when the loss rate is very low. Low loss rate is the most common behaviour we have observed with congestion control. The analytical model could be used to predict the right trade-off between sending FEC or new data.

Other DAG Data-Structures

Finally, even if designed with 3D streaming in mind, the model could be used to other kinds of content based on a Direct Acyclic Graph[6] structure with an equivalent of the quality metric.

[6]Eichhorn proposes in [Eic06] a framework to model the dependencies for generic multimedia data.

Chapter 4

Modeling and Streaming of Natural 3D Scenes

Contents

In the previous chapters we have tackled problems concerning the streaming of 3D *objects*. First with the progressive representation of a particular modeling scheme, and then by the packetization of generic multiresolution 3D objects. In this chapter, we take a higher point of view, and we study the streaming of *3D scenes composed of various 3D objects*. We keep the focus on natural scenes and we use adaptation techniques to improve a global streaming system.

Section 4.1 details what actually *are* 3D scenes, both from a streaming point of view, but also from a rendering and modeling context. Then, in section 4.2, we discuss the current state of the art in streaming of 3D scenes. A first adaptation scheme is proposed in section 4.3. It is based on a classical client-server scheme and deals with the optimization of viewpoint requests on the server. Next, in section 4.4, we study the problem of the deployment of a global streaming system, and provide the design of our next experimental platform.

4.1 3D Natural Scenes

In this section we present a more precise definition of a 3D scene (section 4.1.1). Then we give more details about the different approaches to manipulate a 3D scene given the target application (sections 4.1.2 and 4.1.3). Finally we present the properties of Natural Scenes with respect to applications and regarding other kinds of scenes (section 4.1.4).

4.1.1 A Generic Collection of Entities

From a coarse point of view, a 3D scene is simply a collection of objects representing components of a virtual world. These objects are often called “entities”. Entities may represent actual 3D *solid* objects, but they may also be *lights* (i.e. parameters representing illumination of the scene), fog or wind parameters ... Even geometric objects may be modeled using totally different techniques, for example, a scene may consist in a terrain represented by a height-map (a 2D grid of numbers giving the altitude for each position), buildings modeled using rectangular meshes, trees modeled as L-Systems, etc.

Entities can be only transformations or parametrizations of other *more generic* objects. Those entities are then called *instances* and, hence, they instantiate *models*. For example, we can have a generic house *model*, describing, *heavily* the geometry,

the main colors, etc of a house building, on one hand. And on the other hand, a lot of lightweight *instances* of the house, which are only the actual position of the house and a few deformation parameters (not to have everywhere the "same" house in the scene). This scheme is called *instantiation*, it allows to reuse the content, and hence, lighten the amount of data needed to describe the scene.

Another common usage is to allow entities to be themselves collections of entities. For example, in a natural scene, a *forest* can be modeled as an entity which contains a given number of trees. This scheme, which allows to structure the scene as a *tree*, facilitates common treatments on families of objects. It can also be seen, in an object-oriented modeling way, as a "Composite" design pattern (c.f. [GHJV95]).

4.1.2 From a Rendering Point of View

The main *application* using 3D scenes is the *rendering* process. For rendering purposes, usually, the scene is stored in *more or less* specialized data-structure which most often allows to organize the entities in a *geographical* way (c.f. section 4.3.3.1). Contained entities have specific needs in terms of rendering because they may be modeled very differently; we will not obviously draw meshes with the same algorithms as implicit surfaces. Therefore, the common design idea is *delegation*: each entity provides an ad-hoc rendering function to the upper layer (i.e. the *scene* rendering system); in object-oriented terms, entities provide a "render" or "draw" *method*.

Practical issues make this specialized scene management more complex. As provided by current Graphic Cards and standard Graphics APIs (e.g. OpenGL and DirectX), the most common rendering method is still the *rasterization* process. However, the limitations of this technique induce ad-hoc processing for the scene management. Most entities are stored in a geographical data-structure, but some of them need separate treatment. For example, translucent objects are usually stored separately, because they are not compliant with the *ZBuffer* algorithm. Translucent objects must be checked for visibility and occlusion, and ordered following their viewpoint distance. Another example are dynamic objects; since they may have constant updates of their position, a *geographical* structure is not well adapted to them.

On the other hand, ray-tracing rendering techniques have different requirements. As the main bottleneck for ray-tracing usage is the algorithmic complexity, the scene management should provide adapted data-structures to speed up computations.

4.1.3 Scenes For Streaming

The previous object-oriented delegation of the drawing process to entities, is similarly useful for streaming of 3D scenes. Each entity required to implement representation-aware packetization. In other words, entities provide a

"next_packet" method/function which takes care of the (efficient) binary serialization of the entity. The upper layer (i.e. the scene level) does higher level packetization; for example by prioritizing most visible objects.

The constraint on scene management when considering streaming applications is the adaptation to the viewpoint of users (c.f. section 1.2.2.2). The streaming system has first to implement efficient and accurate viewpoint queries on the scene. We study this issue more precisely in section 4.3. Moreover, in the case where instantiation (c.f. section 4.1.1) has been used to model the scene, the system has to deal with the dependencies between instance and model entities.

4.1.4 Natural Scenes

In our work, we focus on natural scenes, hence we need to consider their specificities.

- First, natural scenes are generally *large*; most often they are *geographically* large, but the concern for a streaming system is size of the data; and in this case, even a Japanese garden can contain very complex models and hence heavy data-structures.
- The second specificity is the *non-uniform spatial distribution* of objects in a natural landscape. Like in the real world, the density of *objects* in a natural environment is very heterogeneous; there can be *clearings*, *dense forest plots*, etc.
- Finally, compared to scenes like urban or indoor environments, natural scenes *suffer* from very few occlusion between entities. This lack of occlusion induces that most often *many* entities are visible at the same time. Rendering and, above all, streaming processes have to interactively manage a heavy collection of visible objects.

We attempt to address these specificities, obviously from a streaming point of view, in section 4.3.

4.2 Current Work

This section provides an overview of related work on streaming of 3D scenes. We have classified the proposed systems into three categories: classical client-server, peer-to-peer (P2P), and video-based systems.

4.2.1 Client-Server Streaming

A system implementing 3D streaming with a client-server design, consists in a host (the server) containing the 3D scene and users (the clients) accessing the scene interactively.

The *ARTE* project (c.f. [Mar00]) is a view-dependent system with server-side selection. The implementation of 3D objects is based on *Topological Surgery*, a progressive compression scheme[1]. ARTE also implements *channel selection*, i.e. based on their loss-tolerance, the pieces of data are sent through UDP or TCP channels.

Effective but *costly* viewpoint adaptation for streaming is presented in [COZ98]. Instead of using classical viewpoint culling techniques (c.f. section 4.3), a coarse ray-tracer is used to compute the visible set of triangles to transmit. This has the advantage of taking into account the occlusion between objects, but unless done very roughly, this technique can not scale to many client because of the computational costs.

Dependency to the viewpoint has also been studied for streaming *single objects*, for example progressive meshes (c.f. [KLK04]) and point-sampled geometry (c.f. [MZ03]). Cheng and Ooi propose receiver driven refinement of progressive meshes which improves the scalability of the server (c.f. [CO08]).

A more scene-oriented proposal is presented in [TL01]. The project implements scene-based walk-throughs. The first idea is the use of *multi-representation* methods: objects are based on meshes but the server is able to dynamically generate low-resolution billboards (images are used as impostors, c.f. figure 2.3). The server also tries to optimize the viewpoint adaptation: each object has a transmission cost, and a view-dependent importance expressing the visual contribution of the object to the scene. Finally to improve the client's interactive walk-through, the server has a *path-prediction* heuristic.

Royan et al. [RGCB07] present a streaming system based on two technologies: multiresolution terrains, and 2.5D city models. The dimension "2.5D" means that most details are represented by textures, i.e. models are very simple geometric objects (e.g. city buildings are only rectangular parallelepipeds, or cuboids), on which precise textures (even *real-world* photographies) are sticked.

The MPEG-4 specification allows the description of 3D scenes, through BIFS (BInary Format for Scenes). For example, Hosseini et al. [HG02] uses the COSMOS framework (c.f. [DG00a]) to stream an animated scene. But the scene is relatively small (6 MB of VRML) and transmission time is negligible compared to the computation and rendering costs.

Other systems have been proposed. [CJD+06] uses a *middle-ware* layer to provide the client with a render-able scene. [SSB04] proposes adaptation to client's resources using various data streams to handle ad-hoc multiresolution. Olbricht and Pralle (c.f. [OP98, OP99]) propose a client-server system based on the RTSP protocol (Real-Time Streaming Protocol) to provide "3D movies" to the user.

Some commercial attempts have emerged too, the most *streaming-based* seems to be "Google-Earth", where the globe *is* a multiresolution spheric height-map. Other objects (buildings, etc.) are not streamed, as they are simple downloadable 3D meshes (represented by compressed XML files). Finally, "Second Life" pro-

[1] c.f. [Tau99] for a state of art of multiresolution compression of geometric mesh-based objects

poses *selective download* of 3D models contributed by the users. It is neither a streaming system nor a P2P system since objects are still stored on (*heavily loaded*) servers.

All these systems provide a more-or-less specialized client-server system with some *local* optimizations. Most ideas could be reused in a streaming system (especially [TL01] or [RGCB07]), as well the ones we present (e.g. in section 4.3.3) since they are mostly complementary.

4.2.2 Peer To Peer Streaming

Peer-To-Peer is a promising target for the streaming of 3D scenes. A first system is proposed in [CBR06] for 2.5D urban environments (extending [RGCB07]).

Another system is *HyperVerse* (c.f. [BHS+08]). It attempts to implement P2P for Distributed Virtual Environments based on HTTP-like peer downloading. This P2P *World-Wide-Web* for 3D models imitates the Torrent networks for large file distribution.

The ASCEND project[2], presented for example in [Hu06, SHJ08], proposes a first P2P-based approach dedicated to 3D streaming. The project starts from a naive implementation and proposes two enhancements. First, instead of using heavy-weight query-response peer selection, an incremental AOI-based (Area Of Interest) strategy is proposed to update neighborhood knowledge. Secondly, a multiresolution management scheme is implemented: considering resolutions *linear* (i.e. we can affect to each resolution an increasing integer), a peer can update information about resolution availability on other peers.

4.2.3 Video-Based 3D Streaming

Instead of transmitting 3D content and letting the client render the geometry, [CBPEZ04] and [NCO03] propose both to render the scene on the server. Given the view-point of the client (sent as a query), the server dynamically renders an MPEG-4 video composition and streams it to the client. This approach may seem non-scalable as the server memory and CPU load is heavily stressed by each different client. But when targeting mobile devices (PDAs or phones) which have limited 3D rendering capabilities, this technique may be the only solution. [LZS+03] proposes a cluster-based rendering farm to scale the server-side rendering. These ideas can be *extended* by designing specialized "Rendering Proxies" to render the 3D data coming from a generic server for mobile devices.

Even if not considered as video, [CYB08] proposes an on demand image-based networked visualization method. The system is view-dependent in the sense that the client *pulls* pre-rendered multiresolution images from a web-server.

[2] c.f. the project's page: ascend.sourceforge.net

4.3 Server-Side Adaptation

In this section we propose a first adaptation scheme improvement for client-server-based streaming of natural scenes. Studying the particularities of natural scenes (section 4.3.1) regarding viewpoint adaptation problems (section 4.3.2), we provide a specialized data-structure to improve the performance and the accuracy of server-side viewpoint culling (section 4.3.3).

4.3.1 Sparse and Heterogeneous Scenes

As explained in section 4.1.1, from a scene-management point of view, the objects can be entities of any model (e.g. points, meshes, procedural plants). For each kind of object, a different encoding, adaptation scheme and packetization approach can be used. To represent geometric objects at the scene level and provide a uniform interface, a common method is to use their axis-aligned bounding boxes, (AABB), the minimal and maximal value of each coordinate of the object (other bounding volumes are sometimes used see for example [BCG+96]).

Moreover, while working with large natural scenes, some characteristics have emerged:

- *Real-life* scenes have, at a coarse level of detail, a two-dimension topology: most scenes (and especially natural ones) are composed of a wide terrain and much smaller objects lying on it.
- The distribution of objects throughout the scene may be very irregular. A scene generally consists in clusters of objects (e.g. a forest is a cluster of trees) and large regions of empty space.

4.3.2 A Data-Structure Problem

We aim at developing an efficient management of a large natural scene on the server: to distribute adapted content to different clients, the server queries the scene for potentially visible objects according to a client's viewpoint. The visible volume is a 6-planes volume called *Frustum* (c.f. figure 4.1). We use the axis-aligned bounding boxes of object as a common structure to perform viewpoint queries on the scene. Thus we need a data structure to manage, at the scene level, a great number of AABB's.

Our goal is to optimize real-time requests on the server and memory footprint: we seek for the most *efficient and accurate* visibility culling traversal of the data structure (in other words, the best selection of visible objects). The creation of the data structure may be costly, and done off-line. For a review on frustum culling of AABB algorithms see [AM00].

Most existing data structures for scenes are driven by rendering considerations: for rendering, viewpoint culling is a less critical optimization than for streaming where the network bandwidth is a considerable bottleneck.

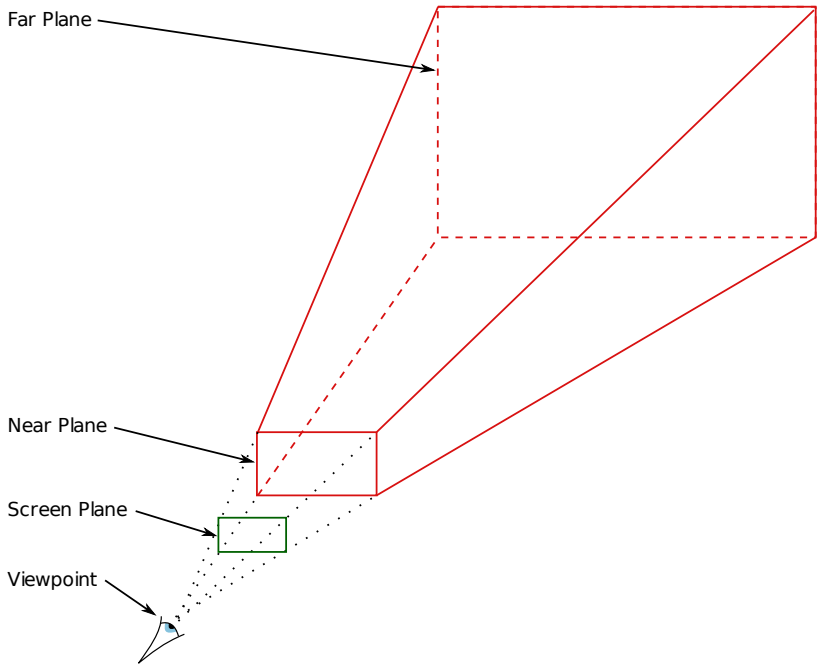

Figure 4.1 – The Frustum representing user's viewpoint.

Moreover, we want to adapt our queries to the characteristics of our scenes. The data-structure has to be optimized for natural scenes. In other words, it has to take into account the distribution of the objects in the scene, which generally is sparse and heterogeneous (i.e. *cluster-based*).

4.3.3 Box-Trees for Wide Natural Scenes

In this section, we propose a specialized data-structure for efficient managing of sparse and heterogeneously distributed scenes. First, we review existing data-structures for scenes management (section 4.3.3.1), then we present our data-structure construction and usage (section 4.3.3.2), and finally we validate through experiments (section 4.3.3.3).

4.3.3.1 Existing Data Structures

This section details data structures classically used for 3D scenes or objects. They fall into two categories: space based (like octrees), and object based (like kd-trees). Section 4.3.3.3 compares the efficiency of our proposed box-trees (section 4.3.3.2) to these structures.

Octrees

In an octree, each node has eight children; the parent bounding box is separated for each direction, cutting in the middle. Octrees can be *sparse* or *plain*:

- With *sparse* octrees we have an adaptive data structure: a space box is divided only if it still contains objects. Unfortunately location-driven accesses to sparse octrees need parsing of the tree which is costly.

- With a *plain* octree we have constant time access to any cell given its location but there can be considerable memory waste as the depth of the tree does not adapt itself to the topology of the scene.

Octrees have quick creation time and provide good results for three-dimensional objects but a global disadvantage of octrees for wide scene management is the lack of discrimination between the three directions of space. As previously mentioned, most "real-life" scenes have a global two dimensions topology[3], therefore all directions should not be treated with the same level of refinement. A simple variant is the use of quadtrees (having to choose 2 dimensions), but they only ease *planar* scenes management. To adapt octree-like structures to surface-based objects in [BHGS06], Boubekeur et al. proposed adaptive octrees which become planar quadtrees when, after some refinements, the surface can be projected to a plane without overlap. But even these octree-like structures do not adapt themselves to a highly irregular distribution of objects.

BSP-Trees and Kd-trees

BSP-trees (*Binary Space Partition*) are binary trees where a region of space is divided in two by a hyper-plane [FDFH90]. Taking arbitrary general hyper-planes introduces computing complexity, and numerical problems if different cutting planes are close. Therefore BSP-trees are often implemented as kd-trees.

A kd-tree is a particular BSP-tree used in databases to store multi-dimensional data so that requests with orthogonal ranges are treated efficiently [BKOS97]. The tree is constructed by dividing the space with axis-aligned hyper-planes for each coordinate successively. The AABB is "cut" on median, in order to get the same number of primitives (which can be seen as n-dimensional points) on each side, and thereby a balanced tree.

Kd-trees have the same drawback as octrees; they are not well adapted to non-uniform scenes (i.e. with privileged directions) as they use a round-robin algorithm (i.e. alternatively x, y, z, ...) to partition the scene. Moreover kd-trees are also not adapted to non uniform distributions; in the case of region which is dense compared to the rest of the scene, most cuts will occur in this region whatever the rest of the distribution of objects is.

[3] we mean *surface*-based topology, not necessarily *planar*

Bounding Volume Hierarchies

Bounded kd-trees (Bkd-trees) [WMS06] and Bounding Volume Hierarchies (BVH) are data structures specialized for bounding boxes. They are used for example for ray-tracing or for collision detection [LAM01].

A Bkd-tree is a kd-tree which stores information about axis-aligned bounding boxes of the primitives in his nodes to allow easier scene updates and object deformations. It keeps the same "round-robin" construction method, so it is suitable for uniform meshes (objects), and designed for ray-tracing with deformable objects (c.f. [WMS06]). BVH are generalized trees of AABB's. There are many variants in the literature (see [Hav04] for a detailed *state of the art*) and BVH are generally classified between R-Trees and semi-R-Trees. R-Trees are generalized multi-valued trees of AABB's where all leaves have the same depth. Semi-R-Trees are R-Trees with variable depth for leaves. (Semi-)R-Trees are usually generated by reducing a *box-tree* which is a binary bounding box hierarchy. In order to create these box-trees several policies can be implemented:

- **cs-box-trees** are kd-trees with data on the leaves. Each parent bounding box is split using the cutting hyperplane.
- **kd-interval-box-trees** are made like cs-box-trees but instead of a round-robin policy for cutting dimensions, the largest coordinate interval of current bounding box is chosen, those are also sometimes called LSF-interval-trees (Longest Side First).
- **(pseudo-)PR-Trees** (Priority-R-Tree) For a n-dimensional dataset, a $2 \times n$ dimensional kd-tree is built with coordinates of bounding boxes: $(x_{1,min}, x_{2,min}, \ldots\ x_{1,max}, x_{2,max}, \ldots)$.

There are also some "extensions" that have been proposed for application-specific constraints, (c.f. [JSLB00, KHM+98, Cam90, GDO00, vdB97]) e.g. the BBD-Trees; which are trees of *"donuts"* (i.e. double bounding/internal boxes), they are used for collision detection in industrial systems.

4.3.3.2 Specialized Box-Trees

Main Idea

Our goal is to define a data structure that allows real-time selection of a set of visible objects of the scene (candidates for immediate streaming). We propose to combine advantages of BVH with adaptation to the topology of "real-life" scenes. We construct binary trees of AABB's by optimizing the *cut*, i.e. the separation of the bounding box of one node in two children bounding boxes. In order to reach our goal we need to:

1. Choose the "cut" direction using the topology of the scene. This avoids the extra work generated by octrees or kd-tree's round-robin for unprivileged directions.

Figure 4.2 – Comparison of the constructions classical data-structures and our proposed one (Dashed rectangles are the cut AABB's). The Octree-like cutting policy cuts the interval by the middle. The kd-like uses a median cut: it leaves as many objects on each side. Both methods cut a lot of objects which lead to many duplicates. Our proposed method minimizes the number of cut objects, and in the same time, gets rid of the empty space between the clusters.

2. Minimize the number of cut objects. As each *stabbed* object will be present in both children, it will therefore increase the number of visited nodes during a request.

3. Try to have the largest "gap" between children. This makes the separation as discriminant as possible and allows earlier culling of parent nodes.

An intuition of the comparison of our construction with the most classical ones is given in figure 4.2.

The Creation Algorithm

These guidelines lead to a top-down creation algorithm, starting from the global bounding box of all object bounding boxes: for a given node, if possible, create

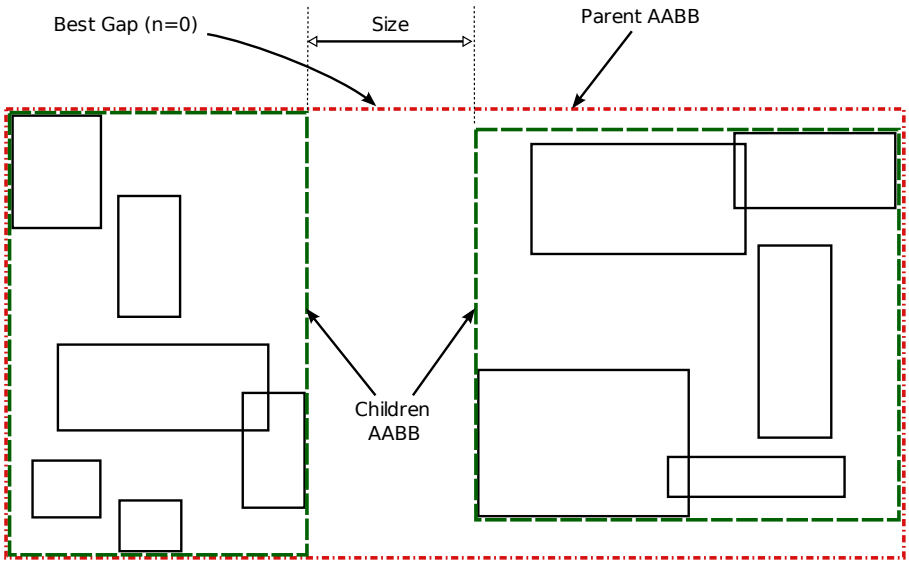

Figure 4.3 – Example of our specialized box-tree construction step.

two children by finding the *best* cutting-gap along every dimension. The following metric is used to compute the *best* gap:

- compute the "size" of the gap s_g and the number n_g of objects that would be cut (i.e. that would be present in both children);
- then (with left-to-right evaluation order):
  $((s_g, n_g)\ better\ than\ current\ (s_c, n_c)) \Leftrightarrow ((n_g < n_c)\ OR\ (n_g = n_c\ AND\ s_g > s_c))$.

Simple 2D usage example of this metric is shown in figure 4.3. The best gap is identified on dimension x as being the longest interval that cuts the minimal number of objects (n=*0* in the example).

The performance of the construction of the tree is not an important issue. Once the tree is built it can be serialized to a file and reused at minor cost, because the scene is considered static (many clients with different time bases can share access to the same content). Therefore, we have implemented a simple algorithm to achieve the construction: at a given level, we project all included AABB coordinates on each axis, and then, search forward the optimal *cutting-gap*.

Viewpoint Query

Independently from the *cutting* algorithm, a box-tree (i.e. a binary tree of bounding boxes) is parsed by a viewpoint query, to retrieve all the objects whose AABB is inside or intersects the view frustum polyhedron.

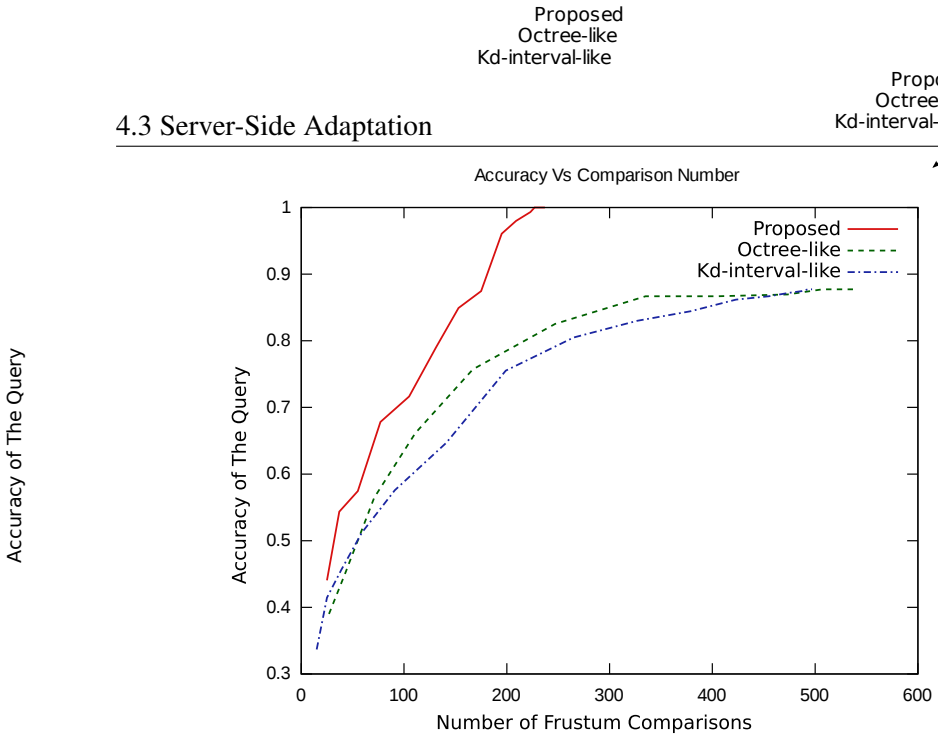

Figure 4.4 – Experimental results comparing the *accuracy* of the viewpoint query regarding the number of frustum comparisons needed to achieve it. With our cutting policy, after 227 comparisons, the algorithm retrieves the exact set of visible objects (*higher is better*).

We need a function which takes view point parameters and an AABB and returns a *flag* (i.e. a "Variant") which states whether the bounding box is totally inside, intersects or is totally outside the view frustum. In our case, we represent the view frustum as six planes (given their Cartesian equation).

Then, we parse the tree using this function. At a given node, we compute the *frustum comparison* for the AABB of the node. If the AABB is totally outside, we reject the whole subtree, if the AABB is totally inside we accept the whole subtree as *visible*, if the AABB intersects the frustum, we continue to query the two children nodes. Since the order is not relevant, the parsing can be equally breadth or depth first, in our implementation we choose breath first.

4.3.3.3 Experiments

We compare our proposed box-trees with classical bounding volume hierarchies constructed with two different *cutting policies*:

- **Octree-like:** a box-tree where, for a given node's AABB, we cut in the middle of the biggest coordinate/edge. We must note that this scheme is *already* an enhancement over classical octrees because this cutting method does adaptation to the *global surface-based topology* of the scenes.

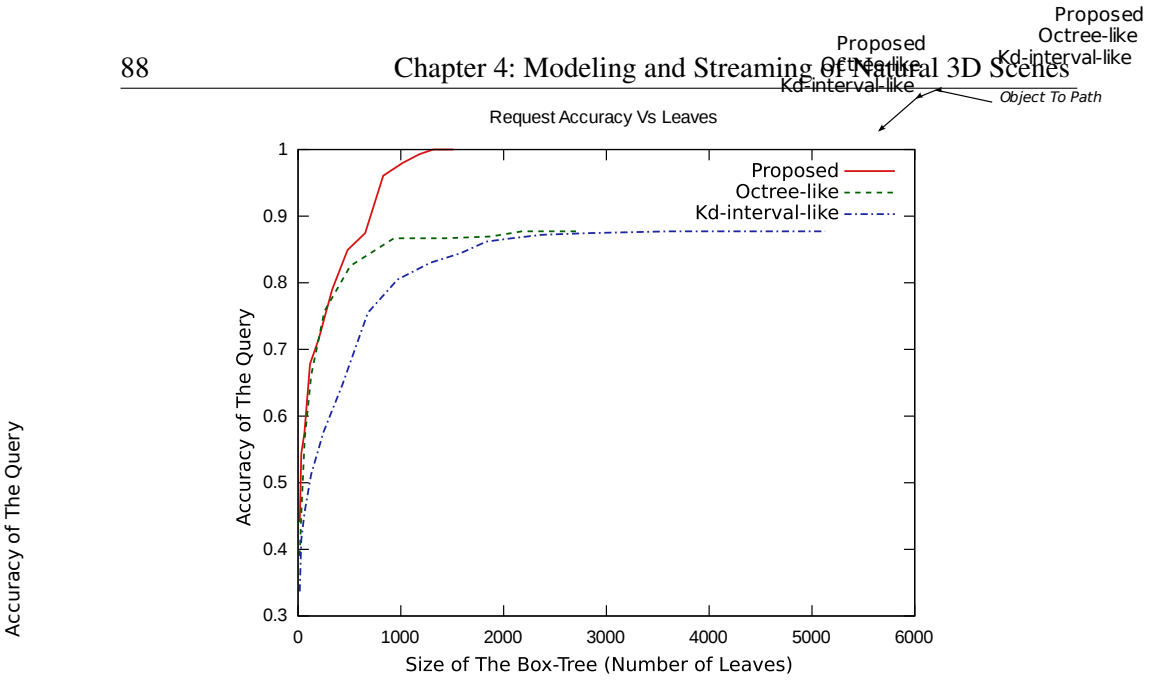

Figure 4.5 – Experimental results comparing the *accuracy* of the query regarding the size of the box-tree data-structure (*higher is better*).

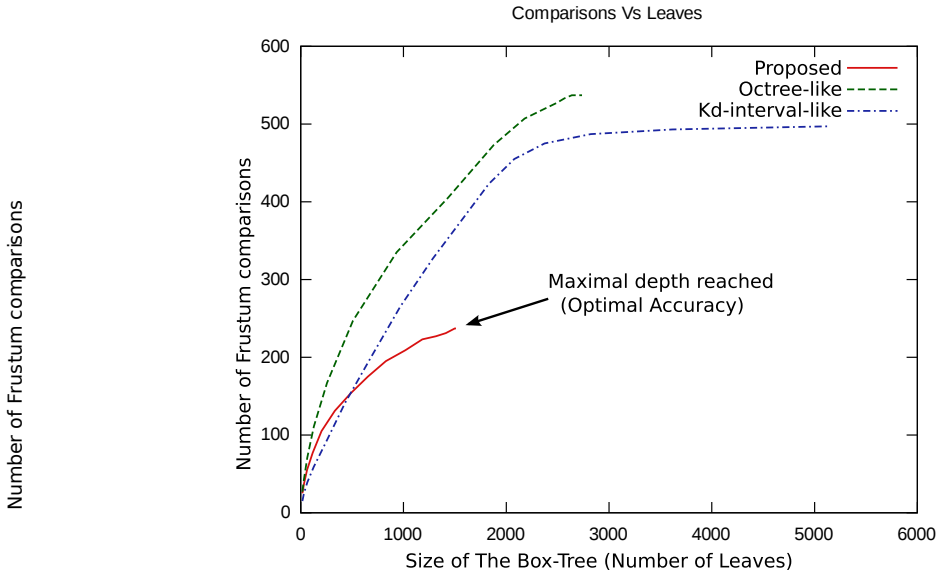

Figure 4.6 – Experimental results comparing the number of Frustum Comparisons needed for the request with regard to the size of the data-structure (*lower is better*).

- **Kd-interval-like:** a box-tree made with a kd-interval-box-tree policy, i.e. a *median* cut on the largest interval which leads to a tree as much balanced as possible.

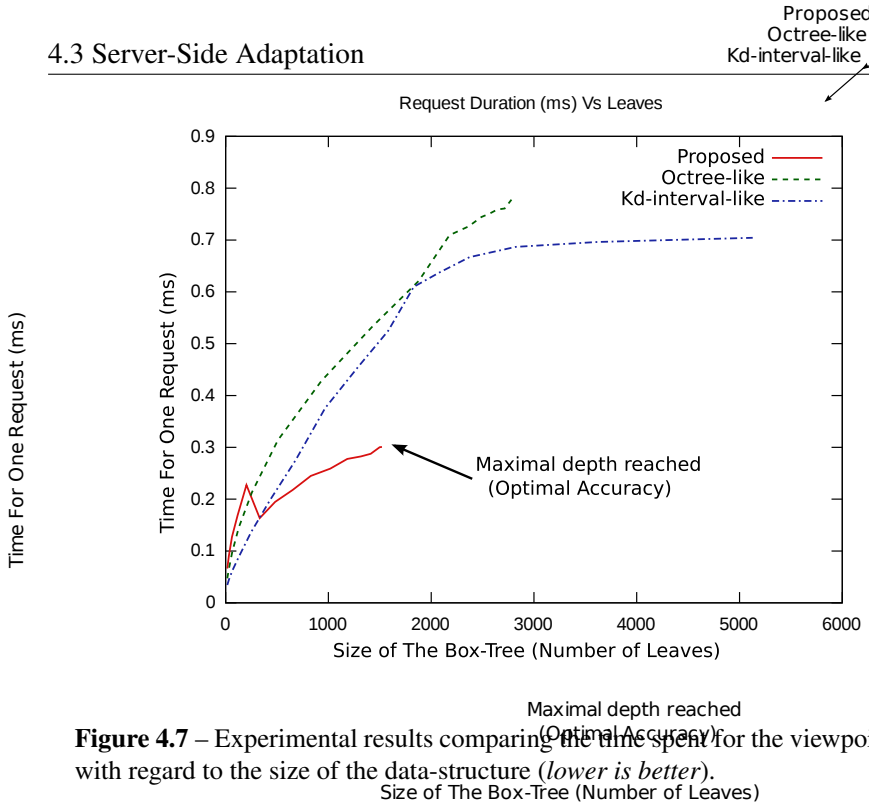

Figure 4.7 – Experimental results comparing the time spent for the viewpoint request with regard to the size of the data-structure (*lower is better*).

We have used randomly generated scenes, with a *privileged* direction, and performed a set of viewpoint requests on it. For each cutting-policy, we generate many box-trees of different sizes; the size of the tree increases the accuracy (or precision) of the viewpoint requests.

For each kind of box-tree data-structure, we collect different variables:

- The **Tree Size** is actually the number of leaves; it represents the complexity of the generated structure.
- The **Number of selected objects** is the number of candidates selected as potentially visible by the viewpoint request. As objects may be cut in various bounding boxes, frustum request can obtain duplicated objects. We also count them and we filter the result to get the exact result. For each scene and viewpoint couple, there is a minimal set of potentially visible objects: the objects whose bounding-box is totally inside or intersects the view frustum. This minimum is the optimal response for a query. Hence, we compute the *Accuracy* of the resulting set as: *MinNumber / RetrievedNumber*.
- The **Number of Viewpoint Comparisons** represents (partially) the complexity of the viewpoint query. Actually, the frustum comparisons (i.e. the function which decides if an AABB, is totally inside, totally outside or intersecting the view frustum) are the computational bottleneck of a request on the tree. We note that the total complexity of the query also depends on the number of retrieved candidates, including the duplicated ones. Particularly, this means that the *Accuracy* of the request improves its response time too.

- The **Response Time** is given as an indication. It is the time spent for a viewpoint request on the data-structure. The executable is Objective Caml code natively compiled for an *Intel Core 2 CPU - 6700 - 2.66GHz* with *4 GB of RAM* running *Debian GNU/Linux, compiled for 32 bits*. This time depends mainly on the number of frustum comparisons and the size of the output (the set of retrieved objects), but, of course, measurements also depend on classical operating system and cache management runtime entropy. Hence, we have run 1000 consecutive requests, and computed the average. Moreover to have a fair comparison, the code managing the request is exactly the same for each data-structure: the difference between cutting-policies only appear during the creation of the box-tree.

We present here a set of results for one generated scene. The scene contains 1500 bounding boxes. We perform viewpoint requests for a a given camera viewpoint. The *exact* frustum culling for the scene retrieves 293 objects (this result is computed separately by testing individually each bounding box of the scene). The results are shown in the figures 4.4, 4.5, 4.6, and 4.7.

Figures 4.4 and 4.5 compare the *accuracy* of the query regarding the number of frustum comparisons for the viewpoint request and respectively the size of the data-structure. It means that our proposed cutting-policy retrieves less non-visible objects for the same number of comparisons. Moreover, on those figures, we can see that our box-trees reach the *100%* accuracy with an acceptable number of comparisons and size of the tree. *Octree-like* and *Kd-interval-like* accuracy curves slow down and both reach the memory limits of the computer before being able to remove all the *non-visible* objects. As mentioned before, to improve the accuracy, we need to increase the depth, and hence the size of the tree.

The figures 4.6 and 4.7 show the performance of the viewpoint requests. About figure 4.7 we note that, from various runs, we can see that there is, obviously some unpredictable entropy for the measured response times, but relative positions of the curves are accurate anyway. Curves for octree-like and kd-interval-like policies are stopped arbitrarily, but the curve of the proposed policy ends when the optimal result is found. The shape of the curve for the proposed policy in figure 4.7, shows worse performance for small sizes of the tree, this means that for non-accurate queries, the size of the returned set is overwhelming compared to the number of frustum comparisons. However, our algorithm outperforms both other competitors when the accuracy becomes bigger than 80%, which is the range that really matters.

Moreover, we shall note that the proposed box-trees have a intrinsic maximal depth, which is the depth where no bounding box gaps can be found. So, at this depth the optimal candidates are retrieved. On the contrary, the other box-trees do not have natural stopping conditions (as there are cases where we can continue cutting forever). For that maximum, the proposed algorithm has 1510 leaves (depth is 20). The first optimal result, for this particular viewpoint query is found “sooner”: when the tree has depth 15 and size 1316, the query retrieves the 293 objects of the exact request (without any duplicate) after performing 227 frustum comparisons

during 0.290 ms. This *maximal depth* explains why curves "stop" sooner in figures 4.4, 4.5, 4.6, and 4.7. Even if possible, making deeper box-trees would be useless since the query is already 100% accurate for all viewpoints with 1510 leaves.

Both other methods need much more leaves, to improve their accuracy. For example, with the presented setup, on one hand, the best result we got with octree-like box-trees retrieves 334 bounding-boxes (with 61 duplicates), i.e. 41 retrieved objects are not actually visible (they are just *close* to the viewpoint). This query requires 2179 leaves, hence, 507 frustum comparisons, during 0.702 ms. If we increase the depth of the tree, we obtain more duplicates, but getting less than 334 objects requires queries of more than 2 ms. On the other hand, the Kd-interval-box-tree performs faster but less accurately than the octree-like box-tree.

4.3.4 Conclusion and Perspectives

In this section, we have presented a data structure optimized for providing quick viewpoint-dependent requests on very large 3D scenes where objects are represented by their bounding boxes. Moreover, for a given tree size, our result minimizes the number of selected objects and thus the amount of data sent over the network. Our box-trees are adapted to scenes which have a global 2D topology and a lot of free space between groups of objects. We have compared them with more classical data structures and results are promising.

Here are some ideas for extending this work:

The metric used to choose the *gap* can be improved: As we want to make the cut even more discriminating at a high-level of the scene, it would be interesting to sacrifice (i.e. cut) some big objects to have a more "balanced" partition of the scene. This would release constraints imposed by larger objects which have their bounding-box including other's smaller ones, for example the terrain or terrain portions.

We can also identify and quantify the possible gain of the proposed method facing the issue of numerical errors. As the cutting algorithm reuses the coordinates provided by the bounding boxes of the objects, and does not make floating point divisions, it does not increase the *floating-point entropy* of the whole scene. This issue becomes critical when rendering is done on a lightweight device which only provides fixed-point arithmetic and sometimes on 16 bits values only (e.g. phones or PDAs).

We could also try to transform our binary trees to more valued trees (like semi-R-Tree to R-Tree conversion (c.f. section 4.3.3.1 and [Hav04]) and evaluate the possible performance gain.

Finally we hope to use those box-trees for dynamic scenes. Work has been done on dynamic update of bounding volume hierarchies (like in [WMS06], [LAM01] and [LAM03]) but our application domain adds heavy constraints: a server must provide a scene for many clients who may not share the same time base.

4.4 Deploying Scalable 3D Streaming Systems

In the previous section we have proposed a data-structure to improve the viewpoint adaptation on server-side. This technique along with some other ones which can be found in the 3D streaming literature (c.f. section 4.2) can be put together in a client-server streaming application (like Wadis 5.2.3). But the scalability of such a system with respect to the number of users and the size of the content, must be addressed to provide solutions implementable in the *real-world*.

In this section we provide the design a testbed system, which allows us to experiment new ideas, for example on scalability, fault tolerance, and adaptation of content to mobile devices.

4.4.1 A Systems' Problem

We now consider *very large* contents, and a large number of online clients accessing them. Clients may have heterogeneous devices.

Among the various challenges one may encounter while deploying interactive multimedia applications (e.g. online worlds), three topics seem important to address:

- The service should be *scalable* to the number of clients. A streaming system should be able to provide access to an increasing amount of users without crashing or imposing heavy latency to the clients.
- The data-structures and the streaming strategies should handle the client heterogeneity. The representations of the content have to be adapted.
- The system has to implement fault tolerance and high availability. In case a software or hardware failure the system should continue to function, at least partially.

Tackling those issues is likely to require the distribution of the content among various nodes of the network. The nodes may be a given number of servers, or optionally the clients themselves; the system may implement a hybrid "multiple clients, multiple servers, peer-to-peer" scheme.

The notion of "session" can not survive in such an hostile environment. It is not efficient to handle user information on server-side in a distributed way. The required communication overhead to maintain coherency would be overwhelming, and thus, not scalable. Therefore one solution is to design systems in which servers are *stateless*. The client application has to become more *intelligent* and manage its own session. In the case of 3D streaming, it means that the client must select himself the subset of the scene it has to render (i.e. the viewpoint adaptation) and request it directly from the involved servers. This scheme is called *"Pull"*-based system because clients pull their content, it is the opposite of the *"Push"*-based systems.

Moreover, distributing the content among various nodes is not a trivial task. For example the amount of *replication* is key issue. A totally disjoint partitioning allows to use the serving resources optimally, but leads to no fault tolerance at all. A whole duplication of the content, allows to be fault tolerant, and to provide to clients maximal ability to choose the most appropriate server (e.g. the least congested). But if the content is too large, having it completely on a server may be a bad resource allocation, moreover in the case of dynamic content, keeping the coherency between highly duplicated contents adds also scalability issues. In particular with 3D streaming, the distribution of scene has also to deal with the dependencies between the entities of the scene, for example, an object on which many others depend could deserve higher replication.

4.4.2 The Context for 3D Streaming

In this section we develop the context and the use case for a scene-based 3D streaming system. The application would target the *real world* with nowadays networks and current hardware resources. For the reasons presented in the previous section, we choose to consider *Pull-based* scheme.

4.4.2.1 Constraints

This streaming system would be constrained by the following issues:

- The content is large, the scene contains many entities represented by different modeling schemes. Even for a single entity, we may use several representations, e.g. 3D models and billboards.
- The users of the system may have different resources. We target client hardware resources from desktop computers to Smart Phones.
- On server-side, we expect to be able to run a server on *commodity* hardware.
- We aim at being able to scale dynamically the system thanks to a variable number of servers. The “new” servers may be clients: a client with strong hardware capabilities and good accessibility *from* the network. The system would hence become an hybrid multi-server / P2P system.
- And the system has to be compliant with 2010’s internet. Servers are the only peers which must be required to have a public and static IP address. Protocols used must be *NAT-compliant*, i.e. they have to be able to cross NAT-networks without extra-configuration (c.f. [SFK08, Sen02]). Users in today’s internet use often local networks which generally stand behind firewalls which block all incoming connexions. For example, some ISPs rent integrated modems/gateways which manage client’s computers as members of a *NAT-ed* LAN.

4.4.2.2 Use Case

The use case from a client point of view of a typical streaming session provides an interactive walk-through.

1. At startup the client application has to know the address of at least one server, it can then initiate the connection.

2. The first transmission is the "map" of the scene, i.e. a data-structure representing the repartition of the objects; spacial repartition in the virtual world, and peer repartition in the network. The system has to deal with the fact that the information about the location of the content may not be totally up to date. If large, this structure can be progressively streamed given the user's viewpoint. From this structure we can start the view-dependent streaming of the entities of the scene.

3. From the *map*, the client application computes a list of objects to retrieve, and optionally assign priorities to them. The list is a visible subset of the map, and *maybe* some prefetched objects (i.e. not *yet* visible). If there are several possibilities, the client also chooses the best server for each entity (the choice-criteria may depend on the bandwidth, the load of the server, ...).

4. The progressive transmission and decoding can then start. The client *pulls* the entities from the selected servers, and decodes progressively any incoming data.

5. When the viewpoint moves the client application has to loop from the third step, optionally by taking into account the temporal coherency of the viewpoint.

4.4.3 Design of a Scalable Streaming System

In this section we describe the design of a 3D streaming system. The goal is to provide a framework for future experimentations on *pull*-based and (hybrid) P2P streaming methods. We aim at experimenting various distribution schemes (section 4.4.4.1) for the content, different control mechanisms (section 4.4.4.2) of the system, the distributed data-structure we call "Map" (section 4.4.4.3), and further ideas like *peer-assisted rendering* (section 4.4.4.4).

As mentioned, the server-side must be a stateless application. A server must reply to simple atomic requests on the content and be generic enough, so that, for example, a client can become a server for the content it has already acquired.

We call the system "Diswa", which stands for *DIStributed WAlk-through*[4].

[4]Diswa can be seen as a complete rewrite of our previous streaming system which was called *Wadis* (*WAlk-through DIstant Scenes*). Wadis is a *push*-based client-server system (c.f. section 5.2.3).

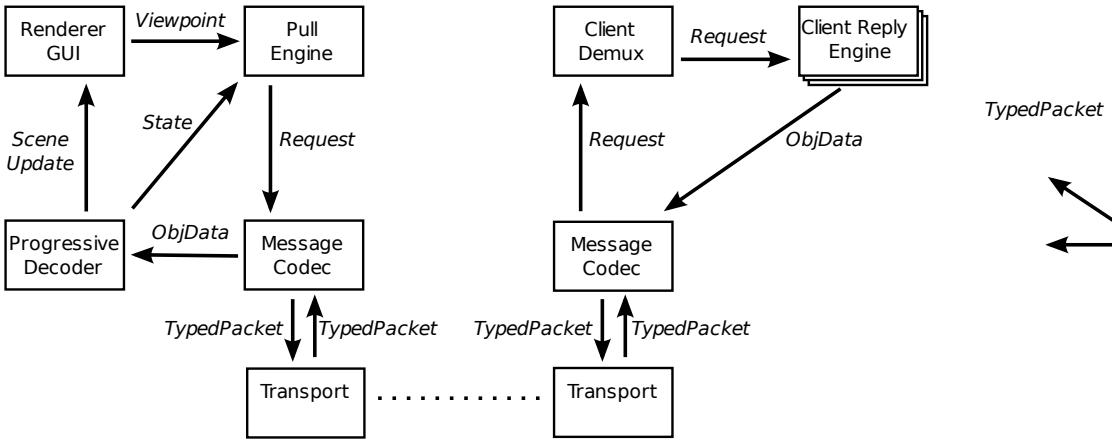

Figure 4.8 – Organization of the components of the client and the server.

4.4.3.1 Requirements

The first requirement is to implement the use case presented in 4.4.2.2, taking into account the constraints presented in 4.4.2.1.

But such an experimental platform must take into account more than its *user-oriented* use case. We aim at performing a lot of experiments in *hostile* conditions (c.f. section 3.5). The system should be for example able to record streaming sessions and replay them. The client-side components should be able to run experiments without GUI[5]; we want to be able to launch various clients maybe remotely on computers which can not handle library dependencies like OpenGL or which will have limited human interaction capabilities. To experiment fault-tolerance, we also require that each component is able to deliberately *crash* or *freeze* on a given control command.

4.4.3.2 System Architecture

The streaming system can be divided in four subsystems:

- *clients* which are the applications run by the users;
- *servers* which deliver the content;
- *controllers* which manage the servers, and the distribution of the scene among them;
- *dispatchers* which are the entry points for the new coming clients, they relay or redirect them to the servers.

[5]Graphical User Interface

The role of the dispatcher can be played by one or more specialized servers. The *controller* is expected to implement the management of the resources and of the distribution content. The exact role of the controller can not be defined now since it will be the *result* of the experimentations performed in section 4.4.4.

The figure 4.8 gives the component architecture for Diswa's client and server subsystems. We detail now the main characteristics of each component.

Renderer/GUI: The *Renderer* executes user's controls on the scene, accepts `SceneUpdate` messages, and sends current viewpoint to *PullEngine* (only if the viewpoint changes). Moreover, the *Renderer* should be the only piece of code which depends on OpenGL & Co. In debug mode, it should be able to move viewpoint sent, independently from actual user's viewpoint.

Pull Engine: The *PullEngine* first tries to obtain the `Map`. Then from user's `Viewpoint` and *ProgressiveDecoder*'s current state, the *PullEngine* generates a `Request` to one or more servers.

Progressive Decoder: The *ProgressiveDecoder* swallows all incoming data from servers, and decodes it as much as possible. From the decoded data, it sends `SceneUpdate` messages to the *Renderer*. The non-decoded data is just buffered (optionally on mass-storage).

Message Codec (Client and Server): *MessageCodec* encodes and decodes messages for other peers. It manages for example client identifiers, or different versions of the applicative protocol.

Transport (Client and Server): The *Transport* component just takes care of "high-level" packetization. It handles, if needed, segmentation and sequencing, channel conversions, and channel choices (if various protocols are used, or if we use various channels for load balancing).

Client Demultiplexer: For a *new* client, the *ClientDemux* component will create a new *ClientReplyEngine*; for an *existing* client, it will just forward the `Request` message to the corresponding *ClientReplyEngine*. By *existing*, we just mean "which already has a *living ClientReplyEngine*," it is not really a *session* as it is temporary and can be lost without repercussions.

Client Reply Engine: The *ClientReplyEngine* executes *stupidly* its incoming `Request`. It dies at the end of the execution of the request, or on a "please die now" request.

4.4.4 Ideas to Experiment

As mentioned previously, the goal of the implementation of Diswa is to be a testbed distributed and peer-to-peer streaming systems. We give now hints about the first experiments we are going to carry out on the platform.

4.4.4.1 Distribution of The Scene

The first issue to solve is the distribution of the content (the entities of the scenes) among the peers involved in its delivery.

Several strategies may be implemented for the distribution mechanism, for example:

- If we try to concentrate users which have proximal viewpoints on the same servers, we can distribute the scene *geographically*, i.e. following the location of the objects in the scene. This repartition does not solve the problem for entities which have no location, for example if we use *instantiation* (c.f. section 4.1.1), the *models* have no location in the scene, the problem is the same for textures which may be used to colorize objects in different places.
- We may also consider a *resource allocation problem* for the *CPU load* of the servers. A dynamic distribution scheme could attempt to equilibrate the resource usage by the servers.
- Another, concurrent, resource allocation issue is the receiving bandwidth of the clients. If we for example try to maximize (while keeping *reasonable* ...) the number of TCP streams held by each client, the total bandwidth may be higher.

The experimental issue will surely be the evaluation of such strategies. We have test the strategies given different criteria: for example, the total CPU load of the servers, the bandwidth provided to the clients, or the level of interactivity perceived by users (c.f. [JW00]).

4.4.4.2 Meta-Management

As stated in section 4.4.3.2, the *controller* subsystem has to manage the whole system in a scalable and fault tolerant way. The service implemented has different aspects which may (or *not*) be independent:

- the *Remote Control* manages the distribution of the scene among the servers (which may be clients);
- the *Coherence Insurance* updates the replicated maps of the scenes detained by the clients and the servers regarding, for example, the server location of the entities;
- the *Watchdog* keeps the system under surveillance to ensure high-availability in case of failures.

The controller has to be himself scalable and fault-tolerant. To achieve this, we have to make it replicable, maybe distributed and as stateless as possible.

4.4.4.3 The Scene Map

The `Map` is a specialized data-structure with which we aim at allowing the client *PullEngine* to take *good* decisions. As the main adaptation scheme remains the viewpoint culling, the map of the scene should handle efficiently the bounding boxes of the entities of the scene (like in 4.3.3), but not only:

- The `Map` must handle the distribution of the content among the peers.
- Lightweight clients may decide more efficiently a streaming strategy if the data-structure contains information about the rendering/transmission costs of the entities.
- There are also *non-located* entities to manage, like models, textures, or lights. Those entities generally bring additional dependencies.
- If the data-structure containing the map is large compared to the bandwidth, it will have to be *streaming compliant*.

4.4.4.4 Peer Assisted Rendering

In section 1.2.2.2, we mention "Multi-model representations". A 3D object, distant from the viewpoint or too computationally heavy for the client device, can be replaced by a low resolution image-based impostor (i.e. a *Billboard*, c.f. figure 2.3).

This technique can be extended to groups of objects. For example, a streaming client using a Smart Phone, could take advantage of a "pre-rendered horizon", i.e. all the objects positioned far enough from the viewpoint could be *replaced* by a rendered image.

In a distributed or P2P environment, those pre-rendered billboards/horizons could be generated by other clients, which have a similar viewpoint. That is *Peer Assisted Rendering*. With commodity graphics hardware, it is now easy to render a scene to a texture object, and retrieve it back to the application. Even if the *rendering assistant* is behind a NAT or a firewall, the computational gain may be worth using a server to relay the images.

Moreover, if the *assistant* is powerful enough, it can compress the images progressively (e.g. *JPEG 2000*, see [MBGB00]). Some recent Smart Phones have hardware accelerated decoding of images and video[6]. The assistant can also for example generate *volumetric textures* for better realism (c.f. [DN04]).

4.5 Conclusion and Perspectives

This chapter has studied 3D streaming from a scene point of view. First, we have presented a data-structure aiming at optimizing server-side viewpoint culling. A streaming server can hence, have more quickly and accurately the visible subset of the scene given the client's viewpoint. Then, we have presented the design of a streaming system targeting internet and allowing us to experiment new ideas about scalability, fault tolerance, and adaptation to mobile devices. Those are not the only perspectives we can foresee.

[6] e.g. the "Hantro" family of chips http://www.hantro.com/index.php?319

Temporal Coherency of The Viewpoint

About viewpoint adaptation, another future direction would be to try to take advantage of the uniformity of visibility queries. During a streaming session, the user translates and rotates his viewing frustum as he navigates, but changes in shape, or especially in size are very rare. Moreover, a classical navigation presents a lot of temporal coherency, which should also be used (like in e.g. [GBP04]). Or more generally, any *path prediction* which would improve the request's response time can be interesting (c.f. [YM06]).

Locality Sensitive Hashing

Casey and Slaney, in [CS06], use Locality Sensitive Hashing (LSH) to retrieve quickly *similar* music samples. If possible, LSH-based hash tables could provide fast-access to approximations of the viewpoint queries. To achieve this, hash-functions sensitive to the geometrical location should be designed.

Pre-fetching

For single 3D objects viewpoint is maybe too unpredictable, and hence pre-fetching has been shown quite useless, for example by Rusinkiewicz and Levoy for the QSplat project [RL01]. But for 3D scenes visited by a significant number of users, pre-transmission of not-yet-visible objects can lead to efficient bandwidth resource allocation. This has been tried in [TL01] with *simple* assumptions. Going further, *most preferred paths* could provide efficient prediction. The predicted paths could for example be inferred from client observation and profiling, such a listening framework would lead efficient pre-fetching like for video streaming (c.f. [GCD02, PCG07]).

Chapter 5

Practical Issues And Lessons Learned

Contents

In this thesis we have tackled the problems with a "systems" and "engineering" approach. All the ideas provided have been tested and validated in applications targeting the *real-world*. Contrary to many industrial projects, a PhD generally benefits from a *fresh* start. This gave us the opportunity to carefully choose the tools that we felt they would lead to safe, easy and efficient software development and therefore validate our theoretical contributions.

In this chapter we deal with the practical aspects of the work we have performed. In section 5.1, we discuss about the lessons we have learned and the tools we have used and evaluated during the whole thesis. Then, we give a recapitulation of the software systems that have been implemented (section 5.2).

5.1 Tools & Lessons

Before entering the battlefield, a PhD student has to *choose his weapons*. We present here the tools and the design ideas that have been learnt, used, and evaluated during these studies.

5.1.1 Operating Systems and Networks

We have first chosen a common operating ground for experimentations. The only platform that has proved mature enough to handle general purpose and especially networked multimedia applications is *UNIX*. *UNIX* must be seen here as the set of standard requirements it consists in, and not, the actual now mostly dead operating system also called *UNIX*. It can also be denominated as "POSIX.1, Core Services" (i.e. IEEE Std 1003.1-1988). This common ground should be the main "dependency" of the software platform as all *modern* operating systems (OS) implement mostly this standard (sometimes using a compatibility layer).

While using the most widely-available *UNIX-like* OS, i.e. GNU/Linux, we had, during this thesis, to depart from our portability requirement:

- We have implemented streaming experimentations using the DCCP protocol (c.f. section 3.5). For now at least, this can be considered, as a Linux-only experimental feature. However, DCCP usage remains optional from the system point of view since there are *portable* fall-backs.

- 3D rendering and client-side user interface, have to take profit from hardware acceleration, from that point of view, and with still portability in mind, we have chosen to rely on OpenGL (mostly early 1.x versions) and SDL (Simple Direct media Layer).

Finally, currently used systems (GNU/Linux, MS Windows and MacOSX) have proved to be not reliable nor performing nor secure. Operating Systems will have to evolve a lot for a brighter future, maybe by acknowledging that Andrew Tanenbaum is right about the benefits of micro-kernel architectures (c.f. [Tan01]). However, from a portability point of view, future OSs will still surely implement some kind of POSIX-like compatibility.

5.1.2 Programming Language

Continuing with Andrew Tanenbaum, we may cite his *first law of software*: *"Adding more code, adds more bugs"* [Tan01]. Programming Languages (and Development Platform) may be the most important choice for a software project. We have chosen the Objective Caml language for our development platform.

Objective Caml is the main implementation of the Caml language, developed by Xavier Leroy et al. since 1985[1]. OCaml unifies functional, imperative, and object-oriented programming under an ML-like type system. It provides inferred very strong static typing (Hindley–Milner), bounds-checking and an efficient garbage collector [Ler90, Ler00, Gar00].

Caml allows the developer to manage very complex problems (c.f. [MW08]) with a few lines of code, very *efficiently*, and, overall, with a lot of compile-time safety:

[1] c.f. caml.inria.fr

- Automatic bounds-checking and strong typing ensure there are no memory unsafe accesses, meaning that the compiled program can not raise a "Segmentation Fault" nor a "Bus Error". Moreover, security holes commonly found in C/C++ programs are natively avoided: there can not be Buffer Overflows (c.f. [AO96, MdR99]).

- There are no "pervasive NULL pointers", hence exceptions like *NullPointerException* common in Java or Python programs are avoided.

- Static typing also means *not dynamic* i.e. classical dynamic duck-typing errors can not occur[2].

- There is no overloading, and no implicit type-casting at all. This feature removes numerical errors present, but very difficult to find, in many programs: e.g. the classical "`2 / 3 == 0`" cast-error.

- Other features provide even more safety, for example, while using `printf`-like functions, there is no reason for a format string to actually *be* a string, types must be different; this inherently avoids "printf format injection" security flaws (c.f. [Scu01]).

The Objective Caml implementation provides a shell-like toplevel interaction loop jointly with bytecode and native compilers (the virtual machine is lightweight and very portable, native code has proven to be very efficient); a debugger (able to do backward-steps !) and support for profiling; a documentation generator; an extensible parser and macro language named Camlp4; build tools; a foreign function interface for linking with C code; and much more ...

To conclude about OCaml, after a few years of heavy usage, and of testing of concurrent platforms, we may state the assessment that Caml is not *the definitive* language and it is not perfect, but it is *by far* the best compromise we can find around. For example, *Haskell* can be considered as the *next step* towards safe programming, but, for now, the lack of libraries, and the unpredictability of the performance are too limiting factors for general usage.

We believe that those common programming methods such as strong static typing, bounds checking, garbage collection, message passing, guided by a safety and security paranoia, are the *way to go* for solving most problems. We should precise that we do not state that "we believe that the future should be like that," instead, we state that "we believe that the past twenty years should have been like that."

We shall finish with a last global empirical advice about programming languages:

Don't look at a language asking what can that language allow a smart and skillful programmer to do. Instead, ask what can that language prevent a stupid

[2] e.g. Python's classical *runtime* exception:
`AttributeError: 'int' object has no attribute 'to_int'`

and lazy programmer from doing. Because programmers are human, so ... essentially ... stupid, and lazy.

5.1.3 Writing and Presenting

The second most used set of tools, for a PhD student, is the one which allows him to write papers and present them: Typesetting tools.

LaTeX is the most popular of those tools, it is may be the first historical attempt to separate the content from the style. LaTeX, thanks to the underlying TeX engine, is also the best tool for the typography of mathematical expressions. That is why it has been pervasively used for this work. But historical reasons, make LaTeX very difficult to use deeply and to extend. A far too permissive compiler, a syntax based on too many special characters, an ugly rewriting-based scripting/macro language, and so on, have driven a lot of people crazy.

There have been a lot of attempts to improve the situation, some of them are still *on top* of TeX, e.g. ConTeXt (wiki.contextgarden.net) or LuaTeX (www.luatex.org), an other try to start from a new base, like Ant (ant.berlios.de) or Lout (lout.wiki.sourceforge.net). ConTeXt has been actively used for typesetting presentations during the present work ... and will be used for its defense.

The last intensively used tool is Inkscape (www.inkscape.org). This application has proven to be the best trade-off for Vector Graphics drawing. Inkscape aims at being very user-friendly while implementing a W3C standard (SVG, Scalable Vector Graphics, c.f. [FJJ03]) to prevent the user from being prisoned by a file format. But its poor internal design and, above all, its C++ implementation make it very unreliable and resource greedy.

5.1.4 Collaborating Safely

We have performed a lot of *worldwide* collaborative work: with *and* from Singapore, with Montpellier and during conferences' journeys. To handle efficiently this distributed team work, collaborative tools have been used. The first tool is the *Wiki*, a simple web-application managing and rendering text files which makes world-wide collaboration very convenient for quick discussion and design.

The second tools are the Version Control Systems. Setting up repositories for code, papers and even presentations has proven very successful. The retained solution has been *Subversion* (subversion.tigris.org). It has been a success particularly thanks to its strong implementation, its integration with other tools, and its easiness to use by newcomers.

Distributed Version Control Systems (DVCS) are more flexible from a development point of view, but they are far less *integrated* and they are more difficult to use by non-initiated people, which makes collaboration harder. For now, DVCS tools are better used on top of classic systems (at least for projects which are not *software-only*), the case used has been *Git* (git-scm.com) on top of Subversion.

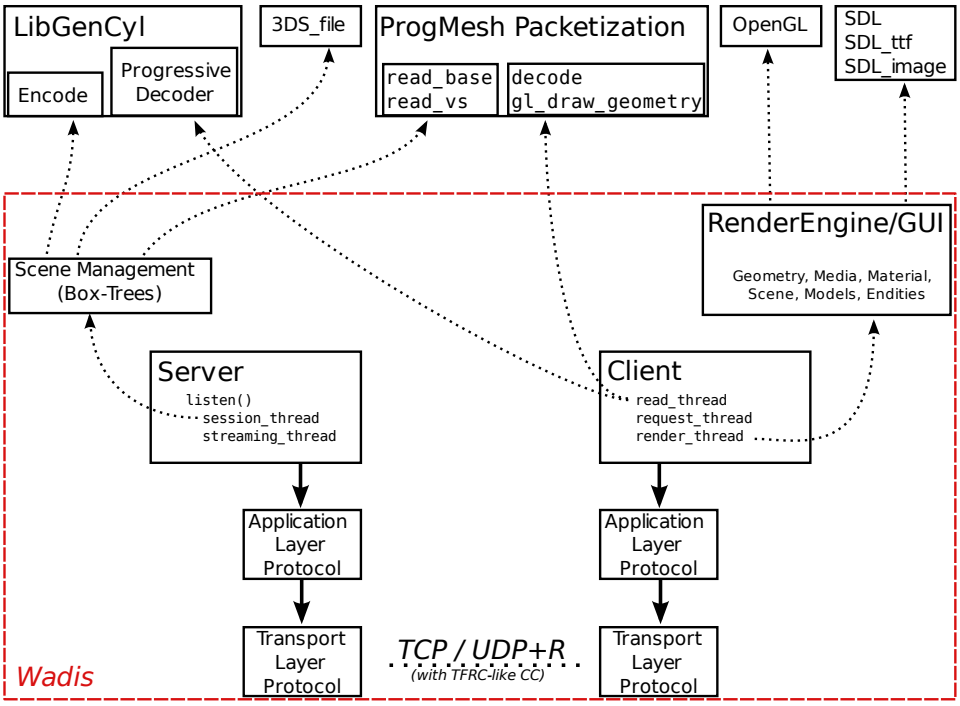

Figure 5.1 – Structure of the components of Wadis.

5.2 Software

Three main software platforms have been developed, *LibGenCyl*, *OMAN* and *Wadis*; they correspond roughly to the chapters 2, 3, and 4.

5.2.1 LibGenCyl

In section 2.5, we present the design of the library implemented the progressive compression scheme. The library plus the tests and the experimentation tools using it, represent 7950 lines of *pure* OCaml code.

5.2.2 OMAN

The *OMAN* tool is described in 3.5.2, it provides traffic generation, UDP tunneling and measurements/statistics. OMAN is also able to "compile" a self-extracting archive including any user provided shell script and the statically linked executable of OMAN itself. This technique allows us to process easy deployment of the experimental setups on distant hosts.

The application code is complex but concise: 1500 lines of OCaml. It relies on a small home-made library called *Unsafix*, which provides generic access to non portable features of Unix/Linux (OCaml's `Unix` standard library *safely* provides

only standard features). We have released *Unsafix* under the MIT license along with some other generic modules we have written: code.google.com/p/yaboon/.

5.2.3 Wadis

In October 2006, the first task has been to implement a naive client-server 3D streaming system, it was Wadis (Walk-through DIstant Scenes). Wadis has been the common *integration* ground for most of the work performed during these last two years (c.f. figure 5.1). Wadis can manage progressive meshes packetized with the Greedy algorithm (c.f. section 3.4.2.2, code mainly developed by Cheng Wei), transmit and render progressively plant models using LibGenCyl (section 5.2.1) as well as importing mesh models using the "3DSMax" file format. Wadis manages the scene on server-side using our proposed box-trees (section 4.3). As mentioned in section 4.4.3, Diswa is now replacing Wadis.

Wadis' client is based on a rendering engine which uses OpenGL and SDL. Wadis client and server core functionalities represent 7000 lines of OCaml.

Chapter 6

Conclusion

Contents

Hopefully the reader is convinced by now that developing a 3D streaming system is extremely challenging. Various attempts have been presented in the literature, but even the best struggle to be widely adopted or used. Unfortunately the work presented here has not completely solved the problem (it did not even try to *completely* solve it), but followed a pragmatic path. Guided by our application (walk-through in a large scene) and the specificities of our data (realistic plants), we started by implementing a naive system. This allowed us to spot challenging issues, choose some of them and propose solutions.

By taking into account the particularities of the content we target, large natural scenes, we have proposed optimizations on some key "adaptation" issues:

- transformation of the content to adapt it to the streaming conditions (section 6.1.1);
- adaptation of the transmission schemes used to make the objects available over the network (section 6.1.2);
- improvement of the viewpoint adaptation implemented on server-side, and preparation of the ground for future work on deployment and scalability (section 6.1.3).

6.1 Contributions

We now recall the main contributions presented in this document.

6.1.1 Plant Modeling

The chapter 2 presented a compact and progressive representation for plant models. This scheme starts from generalized cylinders modeling a plant and outputs a set of interdependent compressed binary chunks. These chunks are ordered thanks to their dependencies and a quality metric so that they can be progressively rendered while decoding. This representation has shown good compression ratio, and can be streamed efficiently since a small subset of the binary chunks provide an acceptable approximation.

Early stages of the work were presented in [MBM+07]. Then, a first version of the whole compression and transmission scheme was published in [MCM+08]. And finally, the current version of the compression method has been submitted to a journal in [MCM+09].

6.1.2 Transmission of 3D Objects

In chapter 3, we dealt with the packetization of multiresolution 3D models over lossy/best-effort networks. We contributed to the development and validation of an analytical model initiated by Cheng Wei and Wei Tsang Ooi at NUS. The model leads to an improved packetization strategy called *Greedy*. The scheme presented in chapter 2 is one example of progressive model that we can packetize using the analytical model.

This work was first published in [COM+07], and then extended for submission in [COM+09].

6.1.3 At The Scene Level

The chapter 4 took *one step back* to consider complete scenes (collections of 3D objects). First, we addressed the viewpoint adaptation performed on server-side: a box-tree data-structure has allowed us to improve the accuracy and the speed of viewpoint culling requests on the scene. This data-structure managing the axis-aligned bounding boxes of the entities of the scene was presented in [MMG07].

Then we considered the deployment of a scalable 3D streaming system. This brought up scalability and distribution issues. Therefore, in order to experiment with distributed streaming, we proposed the design of a testbed application.

6.2 Perspectives

Research in multimedia systems is pluridisciplinary work by definition. Naturally, issues commonly addressed by computer graphics, distributed systems and net-

working researchers need to be handled together. Therefore, perspectives of this work are numerous, since good ideas emerging from these research areas can be useful for bettering a streaming system.

In sections 2.6, 3.6, 4.3.4, 4.4.4, and 4.5, we provide extended perspectives for the different adaptation schemes we have proposed. Among them, we think the next natural step can be the set up of distributed streaming. Indeed, distribution of the scenes and P2P-like schemes like the peer-assisted-rendering are really needed (c.f. section 4.4.4).

On the modeling side, a natural extension of our work would be to enrich tree models by adding leaves. Section 2.6 gives some details. It would also be interesting to experiment with multi-models (meshes, our models, billboards ...). This implies to have a multiple-level progressiveness; at the object level (progressive model of a tree) but also between various representations, from the lightest (billboard) to the heaviest (e.g. very detailed meshes).

Progressive models can also benefit from temporary inference. For example, this means to be able to display branches even sooner then receiving its model (using some sort of inferred model). This may be a good idea, inspired by video coding research.

This thesis proved that our analytical model for streaming is well suited for streaming progressive meshes and our plant models. More generally, we believe our analytical model for streaming can handle any data that can be represented as a direct acyclic graph. There are probably other applications that need to stream large quantities of data, whose data dependencies are naturally modelled by a DAG. This may be the case for progressive models, composed of a base layer and refinement information. It would be interesting to find other applications, to study their data structure and experiment with our streaming strategies.

Navigating inside large scenes require the transfer of large quantities of data and makes the user experience access latencies. Prefetching is a classic solution for reducing these latencies (c.f. [PCG08]). Prefetching can greatly benefit from the natural branching structure of plant model or other models such as respiratory or circulatory systems in human body 3D models.

6.3 Further Work

More generally, some questions raised during these last two years have still no answer.

- Now TCP is becoming ubiquitous on internet; NATs, firewalls and even the implementation of the backbone networks, impeach large scale deployment of any other protocol. Wireless networks are still mainly owned by the telecommunication operators, they can hence adapt the networks to the applications, but is this just a transitional state ?

- The raising of *YouTube* or *DailyMotion* on the internet has been fueled by the availability of content; now, it is easy and relatively cheap for inexperienced users to create/edit/publish audio and video content. The edition of 3D models is still, by far, a difficult and time-consuming task. Who will provide content for 3D streaming applications ?

- Once tools have been provided for designing and implementing efficient 3D streaming systems, the actual large scale deployment must be motivated. What will/would be the *Killer Application* of the 3D streaming ? And that of 3D natural scenes ?

Appendix A

French Summary

Contents

In order to comply with the administrative requirements of the University of Toulouse, in this appendical chapter, we kindly provide a summary of the whole thesis using Molière's language. Dear non-French-speaking reader, please do not worry about this totally redundant chapter, absolutely no additional content will be given there.

A.1 Introduction

La disponibilité des connexions Internet à haut débit pour les particuliers et la puissance du matériel graphique accessible pour un coût raisonnable ont augmenté la popularité des applications interactives d'environnement virtuels distribués (*Networked Virtual Environements*, NVE).

Les NVEs sont l'une des applications pleinement et véritablement « multimédias » de part leur implication de nombreux types de vecteurs d'information : des modèles 3D, des animations, des images, de la vidéo et du son. Ces médias sont des données généralement stockées sur un serveur et celles-ci, collectivement, décrivent un environnement virtuel complet. Un client se connecte au serveur afin de naviguer à travers l'environnement, en requêtant un sous-ensemble des données disponibles sur la base de son point de vue actuel et de ses intérêts. Le serveur

transmet les données au client, qui, au fur et à mesure qu'ils les reçoit, crée une scène 3D partielle qui sera rendue dans un environnement virtuel pour l'utilisateur.

La communauté de recherche multimédia a fait beaucoup de progrès sur la transmission des contenus audio et vidéo de haute qualité. La qualité des objets 3D dans les NVEs est, cependant, encore primitive, fort peu réaliste en général. Des modèles simplifiés ou des représentations à base d'images sont couramment utilisées dans les NVE afin de réduire à la fois les exigences en temps de calcul et en bande passante sur le réseau. Bien que la loi de Moore et, de fait, l'évolution de la technologie des GPU (Graphics Processing Unit) aient fait de la pertinence des préoccupations concernant les exigences de calcul de l'histoire ancienne, les conditions opératoires du réseau restent un goulot d'étranglement. Par exemple, les GPUs disponibles sur le marché des nos jours sont capables de rendre aisément en temps réel la « Stanford's Thai Statue » ; un modèle composé de pas moins de 10 millions de triangles. En revanche le fichier représentant l'imposante sculpture, avec une taille de 122 MO même après compression, demande encore de 1,6 minutes pour être téléchargé, et ce, même sur une connexion très rapide de 10 Mbps. La latence induite par le téléchargement complet d'un tel objet au cours d'une navigation est prohibitive pour une utilisation interactive de l'application concernée.

Parmi les objets que l'on voudrait représenter dans un environnement virtuel, ceux constituant le monde dans lequel nous vivons seraient certainement les plus évidents. D'une part, les communautés de recherche en botanique, biologie et physique acquérissent et stockent d'énormes ensembles de données représentant chaque entité naturelle par un modèle. D'autre part, les utilisateurs désirent naviguer sans encombre dans des environnements virtuels complexes réalistes composés de plantes (arbres, forêts, prairies), de cours d'eau (rivières, ruisseaux, cascades) et de phénomènes atmosphériques (nuages, brume, brouillard).

Dans le présent travail, nous étudions le streaming interactif de grandes scènes naturelles. L'objectif final est de fournir un système capable de permettre à un ensemble de clients indépendants de naviguer de façon interactive dans une vaste scène 3D distante. La scène 3D peut être stockée sur un ou plusieurs serveurs et les clients peuvent utiliser différents type d'appareils (ordinateurs, téléphones mobiles plus ou moins intelligents ou *Smart Phones*). Le réseau qui se trouve entre les participants est un réseau IP « best effort » (comme par exemple l'Internet).

Le présent document présente nos travaux effectués entre octobre 2006 et mars 2009 dans le cadre d'une thèse de doctorat. Cette première grande section, vise à donner une vue d'ensemble du contexte de la thèse (section A.1.1) ainsi qu'une présentation du sujet de recherche (section A.1.2).

A.1.1 Le contexte

Ce travail a été supervisé par Mathias Paulin, Géraldine Morin et Romulus Grigoras à l'Institut de Recherche en Informatique de Toulouse (IRIT) de l'Université de Toulouse dans le groupe de recherche VORTEX (Visual Objects from Reality To EXpression). Wei Tsang Ooi de l'Université Nationale de Singapour (NUS) a

conjointement encadré cette thèse, en particulier pendant une période de trois mois de stage à la School of Computing de NUS ; de juin à août 2008.

Le travail accompli au cours de cette thèse a été essentiellement financé par le projet Natsim (section A.1.1.1) et fait, d'une certaine manière, suite à une précédente étude, que nous avions effectué en 2005 (section A.1.1.2).

A.1.1.1 Le projet Natsim

Le *Nature Simulation Project*[1], financé par l'Agence Nationale pour la Recherche (ANR) avait pour nom de code : `05-MMSA-0004-01`. Il visait à étudier et à fournir des outils de modélisation, de représentation, et de transmission de scènes naturelles à partir du point de vue simultané des communautés d'informatique graphique et multimédia.

Le projet était divisé en cinq groupes de travail, le nôtre étant intitulé *Streaming*. Ce groupe de travail fut destiné à fournir un cadre pour la distribution ainsi que pour la visualisation à distance des scènes naturelles, tout en participant à leur représentation multi-modèle. Ce double contexte nous a donné l'occasion par exemple de collaborer fructueusement avec l'équipe *Virtual Plants* du CIRAD à Montpellier (notamment Frédéric Boudon, Christophe Pradal et Christophe Godin).

A.1.1.2 Streaming de scènes 3D basées points

Cette thèse fait suite en outre à des travaux antérieurs que nous avions effectué sur le streaming de scènes modélisées à base de points. Le cadre était un travail de Master Recherche qui fut préparé au cours de la période allant de Février à Juin 2005. Ce travail est décrit, en français, dans le mémoire « Mise en ligne d'environnements 3D vastes échelonnables : adaptation aux ressources et à la Navigation » (Mondet, 2005)[2], ainsi que dans [MMG05].

Ce précédent travail était fondé sur une architecture client-serveur fournissant un service de navigation interactive dans des scènes 3D distantes dont les géométries étaient modélisées par des *splats*, des éléments de surface circulaires (c.f. figure 1.2).

Les modélisations basées points furent présentées, il y a de nombreuses années par Levoy et Whitted (en 1985, dans [LW85]), mais n'ont gagné une renommée grandissante parmi la communauté d'informatique graphique que plus récemment (c.f. [RL00, KB04, Pau03, BSK04]). Nous avions alors fait le choix des géométries à base de points de part leur intrinsèque tolérance aux fautes ; les maillages étant quant à eux plus fragiles à cause de leurs contraintes de connectivité. Pour gérer les points du côté serveur, notre système était fondé sur une structure de Kd-Tree représentant la scène (pour une présentation complète des Kd-Trees c.f. [BKOS97]). Cette structure nous permit de traiter, sur le serveur, les requêtes de

[1] c.f. www.irit.fr/Natsim

[2] Disponible en ligne : sebmondet.ifrance.com/RapportS3DSebastienMONDET.pdf

point de vue sur la géométrie. Le serveur sélectionnait le sous-ensemble visible de la scène à transmettre au client (comme présenté sur la figure 1.3). Régulièrement le client envoyait son point de vue au serveur (en cas de changement). L'application cliente était fondée sur le moteur de rendu de PointShop3D. PointShop3D est une suite logicielle de modélisation pour les objets basés points, développé à l'EPFL [ZPKG02].

Au niveau du réseau, trois protocoles furent étudiés :

- **HTTP** : le système utilisait un serveur web (Apache) ainsi qu'un script CGI afin de répondre aux requêtes du client (le point de vue était codé dans l'URL) ;

- **TCP** : une serveur TCP spécialisé se montra beaucoup plus réactif et beaucoup moins gourmand en ressources ;

- **DCCP** : le protocole était à l'époque encore au stade expérimental: l'IETF[3] n'avait proposé que des brouillons de la RFC[4], et seule une implémentation en espace utilisateur était disponible.

À la fin du travail de master, à la fois les systèmes client-serveur fondés sur HTTP et TCP avaient été efficacement implémentés en C++. Le système fondé sur DCCP n'avait en revanche guère atteint un état stable avant la fin du temps imparti.

Ce travail fut le premier contact de notre équipe avec le streaming 3D.

A.1.2 Streaming de scènes 3D naturelles

Comme le projet Natsim nous l'eut défini, notre mission a été de mettre à disposition de vastes scènes 3D naturelles pour une visualisation distante. Nous avons ainsi pour objectif de permettre à des utilisateurs sur un réseau de faire l'expérience d'une navigation interactive par l'Internet ; les clients pouvant quant à eux disposer de ressources hétérogènes et les conditions du réseau étant variables.

Les scènes naturelles sont modélisées avec des modèles spécifiques et ont une structure particulière. Des modélisations adaptées pour les plantes [Blo85], pour les cours d'eau [YNBH09] ou pour des nuages [BNM+08] ont été proposées. Notre objectif est de conserver autant que faire ce peut la cohérence botanique et physique de la scène.

Nous donnons maintenant quelques exemples de domaines d'application de notre thème de recherche (section A.1.2.1). Ensuite, nous fournissons aperçu détaillé des problématiques imposées par le streaming 3D appliqué aux scènes naturelles et de l'idée directrice menant à leur prise en charge (section A.1.2.2).

[3] Internet Engineering Task Force
[4] Request For Comments

A.1.2.1 Applications ciblées

La première application qui motiva notre travail vint directement du projet Natsim. La communauté de recherche en infographie botanique, met en place une énorme quantité de données résultant de la simulation de la croissance des forêts et des environnements naturels. Le groupe de travail en charge du rendu en temps réel des modèles fournit des outils de visualisation des scènes, issues de ces simulations. Le streaming 3D est entre les deux. L'outil de visualisation doit être une application cliente de la visite distante interactive de ces environnements virtuels.

Certains jardins botaniques de part le monde offrent déjà une base d'images, donnant forme à une visite en ligne. Par exemple, le site Web « Découvrez les jardins de Kew »[5] propose une visite officielle des « Royal Botanic Gardens » à Kew, près de Londres, sous forme par exemple d'images panoramiques. Une immersion totale dans un environnement virtuel du jardin botanique serait une belle amélioration de l'existant.

Une autre application de la cohérence botanique sont les jeux éducatifs axés sur la nature. Ces jeux en ligne, par exemple, proposeraient une chasse au trésor sur l'Internet fondée sur les énigmes botaniques et biologiques.

Enfin, l'adaptation à des dispositifs hétérogènes utilisateur peut fournir des outils pour les travailleurs mobiles, comme les architectes paysagistes, en utilisant un PDA (*Personal Digital Assistant*) Un rendu du paysage potentiel simulé pourrait être fourni à l'utilisateur. En outre, il serait intéressant de mettre en relation le point de vue virtuel et la position réelle de l'appareil. Cette application utiliserait par exemple, la puce GPS (*Global Positioning System*), l'accéléromètre et/ou l'appareil photo qui sont intégrés dans les PDA et les téléphones modernes.

A.1.2.2 Problématique et solutions génériques

Quatre problèmes principaux

La figure 1.4 présente un système simple client-serveur de streaming 3D. Notre cas d'utilisation est le suivant : les multiples clients ayant des dispositifs différents, se connectent au serveur afin de visualiser une scène 3D, que le serveur transmet progressivement. Les principales questions soulevées sont indiquées sur la figure.

- **Contenu très volumineux :** Naturelles ou non, les scènes peuvent être composées de centaines voire de milliers d'objets 3D. Même un simple objet peut être gigantesque en termes de stockage. Par exemple, la géométrie de la statue de David, du « Digital Michelangelo Project », représentée par un maillage, se compose de 2 milliards de polygones ; même après compression, la taille du fichier est de 32 GB. Le téléchargement complet de ce type de modèle avant la visualisation est déjà complètement prohibitif.

- **Dispositifs clients hétérogènes** : Aujourd'hui, les clients des applications sur l'Internet peuvent avoir des dispositifs très hétérogènes : Ap-

[5] c.f. lien www.explore-kew-gardens.net

pareils comme les assistants numériques personnels (PDA) ou les téléphones mobiles sont maintenant en mesure de rendre des scènes 3D simplifiées. Les fabricants de matériel fournissent maintenant des puces de rendu ayant de bonnes capacités, comme le chipset NVIDIA GoForce qui prévoit l'accélération du Rendu 3D depuis le modèle 4500. D'autre part, ces petits dispositifs peuvent parfois ne pas avoir d'arithmétique flottante. Par exemple, l'architecture ARM (Acorn RISC Machine), qui est la plus largement utilisé sur les mobiles, est un processeur 32-bits RISC qui ne prévoit l'arithmétique flottante qu'en option (e.g. l'extension VFP).

- **Interactivité :** Quel que soit le dispositif qu'il utilise, le client doit être autorisé à naviguer dans la scène de manière quelconque. Cela signifie que le point de vue du client est toujours en évolution et est le plus souvent imprévisible. C'est la principale différence entre le streaming 3D et de streaming vidéo. Pour la 3D, même si la position du spectateur peut être considérée continue (si l'on considère les « sauts » dans l'espace relativement rares), une petite rotation du point de vue peut impliquer plusieurs « kilomètres » pour l'horizon visible. L'ensemble des objets visibles peut varier de façon spectaculaire. D'autre part, le point de vue d'une vidéo n'a qu'une seule dimension : le temps. Pour la plupart des usages du streaming vidéo, le spectateur va voir la vidéo en continu, même si des systèmes peuvent gérer des sauts dans la séquence vidéo, il peuvent être considérés comme des cas exceptionnels (c.f. [LGS+00]).

- **Les conditions du réseau :** Le terrain commun, ô combien dangereux, de tous les systèmes distribués, est le réseau. Nous considérons que l'Internet, le réseau de réseaux, est de notre base opérationnelle. Cela signifie que nous devons prendre en compte des conditions variables et que nous n'avons droit à aucune garantie de qualité de service.

Un mot-clef : L'adaptation

De nos jours, nous ne pouvons guère concevoir une application multimédia distribuée ou produire des contenus multimédia sans avoir en permanence l'hétérogénéité et le contexte de utilisation à l'esprit. L'hétérogénéité signifie que nous devons prendre en compte des contraintes qui sont connues à l'avance (les dispositifs, les réseaux, les exigences de l'interactivité utilisateur). Le contexte d'utilisation signifie qu'une application multimédia doit s'adapter, au moment de l'exécution, à un contexte en pleine évolution (e.g. conditions variables du réseau, la charge des serveurs, la situation géographique d'un utilisateur, etc.). L'adaptation aux contraintes prévisibles ou imprévisibles ou à d'autres facteurs est donc primordiale pour les systèmes multimédia distribués modernes. Nous détaillons maintenant les différents aspects de l'idée générale d'adaptation en tant que solutions à nos problèmes.

- **La compression** : La première *solution* est de transformer les données, i.e. adapter la scène 3D à nos besoins. Étant donné que le contenu peut être très grand, et qu'il doit pouvoir passer dans un tuyau fort mince, notre premier objectif est de représenter les contenus de manière aussi compacte que faire ce peut.

- **La progressivité** : Un autre prétraitement, est celui de faire subir au contenu une transformation le rendant progressif (ou multi-résolution). Par progressivité, on entend que le contenu doit être organisé afin d'être partiellement décodable ; une première partie d'un flots de données progressif doit permettre le décodage d'une grossière approximation de l'objet modélisé (i.e. basse résolution), et les parties suivantes améliorer la qualité de l'approximation (i.e. résolutions supérieures).

- **Paquétisation adaptée** : Du point de vue d'une application multimédia, une transmission sur le réseau est soit un *flux*, i.e. une séquence ordonnée d'octets (utilisation de protocoles comme TCP), soit une série de paquets plus ou moins indépendants. Un paquet peut-être perdu dans le cas de l'utilisation de protocoles comme UDP. Dans le premier cas, les systèmes d'exploitation ont la tâche de cacher les pertes et les désordonancements provoqués par le réseau sous-jacent à l'application, cela diminue les performances brutes. Dans le second cas, la flexibilité permet l'optimisation des performances, mais l'application doit gérer les difficultés induites par le réseau. Par conséquent, l'on doit faire face au problème par l'adaptation des techniques de transmission aux conditions du réseau.

- **Le préchargement** : L'interactivité et, par conséquent, les variations du point de vue, ne sont point obligées de condamner le concepteur de systèmes de streaming 3D à mettre en œuvre uniquement des systèmes *réactifs*. Le préchargement est une technique d'adaptation *proactive* pour les applications multimédias. Elle consiste à transmettre des données qui ne sont pas encore visibles à l'avance afin de profiter aux mieux de la bande passante, surtout pendant les « temps morts ».

- **Les représentations multi-modèle** : Enfin, un dernier système d'adaptation est la *multi-modalité* des objets 3D. L'idée est de posséder plusieurs représentations d'un même objet, et d'adapter leur utilisation au réseau, à l'utilisateur, à son point de vue, ou aux capacités de sa machine. Par exemple, une plante dans une scène naturelle peuvent être représentés (et transmis), comme un modèle à haute résolution lorsque l'utilisateur l'inspecte de près. Mais le système peut choisir d'envoyer uniquement une image fondée sur la représentation en basse résolution (appelée *billboard*), lorsque le client est loin ou que ses capacités de rendu ne permettent pas à son dispositif d'affichage de rendre suffisamment de géométrie.

A.1.3 Le reste de l'annexe A

Les travaux exposés dans le présent document visent à contribuer au streaming de scènes 3D naturelles, en proposant des mécanismes d'adaptation. Dans la prochaine section (A.2), nous développons notre étude sur un représentation progressive des modèles de plantes qui sont parmi les plus importants objets 3D des scènes naturelles. Puis dans la section A.3, nous présentons une méthode de paquétisation adaptée au contenu 3D multi-résolution, et nous l'appliquons à notre modèle progressif pour plantes. Nous présentons nos travaux sur de grandes scènes, sur l'adaptation du système à l'interactivité du point de vue du client (section A.5). La section A.6 fait état des questions des leçons apprises en ce qui concerne la pratique et la mise en œuvre des solutions proposées. Enfin, la section A.7 conclut et donne des perspectives sur le sujet.

A.2 Représentation progressive de modèles de plantes

Dans cette section, nous proposons un système de compression progressive de plantes fondé sur des cylindres généralisés. Cette représentation, par son aspect multi-résolution, est bien adaptée à une utilisation dans le cadre d'une application de streaming. Elle nous permet de paquétiser, de transmettre et de rendre les plantes avec une amélioration progressive de la qualité.

A.2.1 Pertinence et spécificités des modèles de plantes

Les plantes sont des objets fort importants dans un monde virtuel. Tout comme dans le monde réel, les plantes contribuent à créer un cadre agréable et réaliste environnement de réalité virtuelle, en particulier ceux qui impliquent des scènes naturelles. Une modélisation réaliste des plantes dans les NVE est par conséquent cruciale dans des applications telles que les forêts virtuelles ou les visites jardins botaniques, où les utilisateurs sont (ou *seront*) amenés à inspecter les végétaux de près et, éventuellement, d'interagir avec.

Toutefois, des modèles végétaux réalistes et détaillés peuvent exiger des centaines de milliers de primitives si modélisés avec de classiques surfaces polygonales. Remolar et al., dans [RCB+02], estiment qu'une plante générée par Xfrog, célèbre plate-forme de modélisation de plantes[6], peut être composée de plus de 50 000 polygones pour représenter les branches. Les plantes peuvent avoir plus de 20 000 feuilles, qui se composent de polygones. Neubert et al., dans [NFD07], ont rapporté avoir utilisé des modèles constitués d'un maximum de 555 000 polygones. Ces chiffres sont pour une seule plante. Dans des scènes naturelles, telles que les forêts, l'on doit s'attendre à trouver un très grand nombre de plantes. La taille de ces plantes motive la nécessité de les diffuser progressivement, plutôt que d'attendre qu'elles soient complètement téléchargées avant d'être affichées. La progressivité est ainsi motivée par des contraintes de performance : la bande passante

[6] http://www.greenworks.de

du réseau, mais aussi dans la taille de la mémoire, la distance de la plante au point de vue, etc.

Les représentations progressives modélisant des objets 3D génériques ont été intensivement étudiées. Par exemple, des techniques de compression multi-résolution ont été appliquées au maillages triangulaires (c.f. [Hop96, AD01, AG05, Tau99]), aux surfaces par points (c.f. [Pau03, RL00, KB04]) ou même à des représentations hybrides (c.f. [CN01]) Toutefois, ces représentations ne sont guère adaptées aux plantes, en raison de la structure topologique de leurs branches. Par exemple, avec des maillages progressifs, il devient difficile de supprimer les triangles au-delà d'un certain niveau de simplification et, par conséquent, la représentation des plantes par des maillages progressifs ne donne pas des résultats satisfaisants (c.f. [RCB+02]).

La figure 2.1 montre que la simplification d'un maillage représentant un arbre ne conserve pas la topologie de celui-ci, et en particulier sa connectivité. Ainsi, des représentations progressives adaptées à la topologie de plantes sont nécessaires. Notre objectif est de fournir une représentation progressive et compressée pour les modèles de plantes, qui préserve leur *cohérence botanique*. Nous voulons, en effet, nous assurer que la connectivité entre les branches à chaque étape de décodage est préservée et, si possible, que le réalisme de la forme des branches est en accord avec l'espèce des plantes.

A.2.2 Représentation et modélisation des plantes

Les travaux antérieurs ont porté sur la façon de bien modéliser une plante (c.f. [RCB+02, Blo85, PMKL01, PL90, NFD07]) ou comment créer facilement une plante dans un environnement virtuel comme l'a proposé, par exemple, le projet *Dryad*[7]. La géométrie des plantes est particulièrement complexe et a donc motivé une série de représentations dédiées à des besoins spécifiques (c.f. [DL05, BMG06]), où les branches et les feuilles sont généralement traitées séparément.

A.2.2.1 Modélisation procédurale: les L-Systems

D'un point de vue de la modélisation, une coutume bien connue est le système de modélisation des plantes par *L-Systems* (c.f. la figure 2.2). Les L-Systems ont été introduits et développés en 1968 par le théoricien botanique et biologique hongrois de l'Université d'Utrecht, Aristid Lindenmayer (1925-1989). Un L-system est constitué d'une chaîne de caractères représentant la structure de branchement par une grammaire formelle (c.f. le livre de Prusinkiewicz et Lindenmayer : [PL90]). Les règles de réécriture de la grammaire permettent de simuler la croissance de la plante. La topologie et la géométrie de la plante sont données par une tortue de style logo qui interprète les symboles de la chaîne géométrique an tant que commandes de dessin (c.f. [Pru86, FKMP03]). Nous notons que, dans ce système, la géométrie d'un symbole est construite en fonction de la géométrie des éléments

[7] Dryad: http://dryad.stanford.edu

précédents, les feuilles sont des instances à différents endroits d'un même symbole géométrique.

A.2.2.2 Surfaces paramétriques et implicites

L'idée précédente a inspiré beaucoup de travaux (dont le nôtre d'une certaine manière). Plus généralement, des représentations de haut niveau pour les branches ont été proposées ; soit sur la base de représentations paramétriques (c.f. [Blo85]) soit à partir de surfaces implicites (c.f. [GMW04]). Elles s'appuient sur un squelette de branches qui est étendu par un rayon (donné par les sections ou les fonctions implicites). Le squelette est défini comme un ensemble de courbes paramétrées interconnectées. Ces représentations de la structure topologique ont l'immense avantage d'être compactes par rapport à des représentations plus discrètes telles que les maillages et fournir un support pour l'animation (ce qui n'est pas le cas des modèles simplifiés, dont la connectivité est perdue, par exemple, dans la figure 2.1). Par défaut, cependant, ils ne sont pas adaptés à une description progressive. Notre objectif dans ce chapitre est précisément de combler cette lacune.

A.2.2.3 Le rendu des plantes

Du point de vue du rendu, quelques representations sont basées sur les *billboards* ; i.e. « panneaux » pré-rendus sous formes d'images utilisés comme imposteurs, c.f. [MNP01, DN04, BCF+05] ; voir aussi la figure 2.3. D'autre part, des représentations basées sur des points (c.f. [WP95, DCSD02]) ou des polygones (c.f. [RCB+02, ZBJ06]) ont été proposées, après adaptation pour l'affichage des arbres. Ces précédentes études se concentrent principalement sur le feuillage (feuilles) et peuvent donc être considérées comme complémentaires à la nôtre, car elles sont généralement complétées par des représentations polygonales du tronc et des branches. Si ces représentations offrent des résultats intéressants, elles nécessitent habituellement une grande quantité de données, en particulier les points et les images.

A.2.2.4 Les cylindres généralisés

Les cylindres généralisés sont une représentation se concentrant sur la structure des ramifications de la plante, i.e. « basée squelette » (c.f. [Blo85]). Une branche est représentée par un axe, une courbe paramétrée définie par un ensemble de points de contrôle, ainsi que des paramètres définis le long des branches (figure 2.4). Ces représentations génériques de haut niveau peuvent alors être affichées comme des cylindres généralisés (c.f. [Blo85, PMKL01]) ou des surfaces implicites (c.f. [GMW04]) et sont beaucoup plus compactes que la représentation par un maillage. Des travaux récents ont également étudié le rendu en temps réel de cylindres généralisés à l'aide de matériel graphique moderne (c.f. [GM03, BW05]). En outre, cette représentation, qui est basée sur un squelette de la structure, peut être étendue avec des informations cinétiques/dynamiques pour faire de l'animation.

Les branches sont organisées dans un arbre n-aire ; structure de données modélisant la structure de la plante. Nous appelons une telle structure de données un *n-arbre*, afin d'éviter toute confusion avec le type de plante « arbre », qui est l'objet de modélisation. Cette représentation par cylindres généralisés a été choisie comme point de départ de notre travail (voir aussi [Bou04]).

A.2.3 Une représentation progressive de modèles de plantes

Dans cette section, nous détaillons le développement de notre représentation et son codage binaire. Ces travaux ont été publiés pour la première fois dans l'article [MCM+08], puis étendus dans [MCM+09].

La figure 2.5 décrit les étapes du codage à la distribution de notre représentation.

A.2.3.1 Données en entrée

Notre représentation se concentre sur la structure des ramifications de la plante et est ainsi fondée sur son squelette. Chaque branche est un cylindre généralisé: un axe, qui est une courbe de Bézier 3D définie par ses points de contrôle[8] et des paramètres *axiaux* tels que le rayon, la couleur ou la texture modélisés par des courbes de Bézier fonctionnelles le long de la branche. Dans la pratique, on utilise seulement les rayons en tant que paramètres axiaux 2D. Les branches sont organisées dans un arbre n-aire, structure de données donnant la structure topologique de la plante. La racine du n-arbre est le tronc de la plante et les branches portées par le tronc sont les fils de ce tronc dans le n-arbre. Chaque fils contient un « paramètre d'attachement » ($u \in [0,1]$) donnant la position du premier point, où la branche s'attache sur sa branche parente (comme dans [PMKL01]).

Pour les paramètres axiaux, nous prenons l'exemple du rayon, mais le système peut être adapté à des textures ou des couleurs. Le cas du rayon de la branche illustre la manière dont les attributs le long de la branche sont codés. Un rayon est défini comme une valeur réelle le long de la branche ; une courbe de Bézier (u, r).

La figure 2.6 récapitule la structure du modèle choisi.

Afin de tester nos algorithmes et méthodes, nous avons utilisé deux modèles de plantes numérisés par Sinoquet et al. [SRG97]) et par Costes et al. [CSKG03]. Nous avons aussi généré des modèles de plantes à partir de L-Systems pour compléter notre jeu de tests.

A.2.3.2 Processus de décorrélation

Afin de mettre en place une représentation multi-résolution compressée de nos plantes, nous exploitons l'auto-similarité des courbes de Bézier, représentant les branches et les rayons séparément. L'idée de l'algorithme de compression est de remplacer la valeur absolue de codage de la plupart des points de contrôle par

[8] Pour une définition extensive et des courbes de Bézier, c.f. [Far02]

des différences à un petit ensemble de courbes *moyennes*. Nous regroupons les branches et les rayons de façon autonome pour profiter de la similitude dans chaque groupe de courbes de Bézier, afin d'avoir différences faibles, donc très corrélées. Par conséquent, celles-ci sont codables avec un moins de bits, ce qui conduit à un codage plus compact.

Un aperçu simplifié du processus de décorrélation est montré dans la figure 2.8 pour le cas des courbes de Bézier représentant des branches (le processus pour des rayons est équivalent, mais moins visuel).

La normalisation

Afin de comparer, et de coder des différences entre, deux branches, nous avons besoin d'une représentation standard des courbes de Bézier. Cette normalisation repose sur deux étapes. La première est facultative, mais applicable à chaque courbe de Bézier qui a un degré supérieur à 2. La seconde, lui obligatoire, est différente pour les branches et pour les rayons afin de profiter de leurs propriétés géométriques. La figure 2.9 (partie bleue, à gauche) montre comment ces étapes sont enchaînées.

Pour avoir des courbes de Bézier comparables par leurs points de contrôle, nous effectuons un prétraitement sur la courbe : nous utilisons un algorithme de réduction du degré (c.f. figure 2.10). Dans la pratique, toute courbe de Bézier de degré plus grand que 2 est approximée par une courbe de degré 2.

L'étape suivante dite de *normalisation*, se différencie selon le type de courbe :

- **Pour les branches :** Nous rendons toutes les branches comparables grâce à une transformation affine définie de manière à ce que les premier et dernier points de contrôle de la courbe de Bézier, se déplacent respectivement à l'origine (0,0,0) et au point (0,0,1) (c.f. figure 2.11). Il s'agit d'une translation 3D, de trois angles de rotation autour des axes et d'un facteur d'échelle uniforme.

- **Pour les rayons :** Il s'agit uniquement d'un facteur d'échelle.

Le groupement des courbes

Le regroupement des branches est une étape dans le processus de décorrélation qui influe sur la performance ensemble du système. La précision de l'approximation par des courbes de Bézier *modèles*, ainsi que les performances du codage entropique des détails dépendent de la qualité du groupement.

Le groupement est une fonction globale qui prend en argument des courbes de Bézier et qui renvoie un ensemble de groupes de courbes de Bézier. Cette fonction peut être considérée comme une série de *filtres en cascade*, ou algorithmes de groupement. Nous avons mis en place plusieurs filtres de groupement, chacun vise à répondre à différents critères :

- l'efficacité de compression ;

- la réduction des erreurs de quantification ;
- l'aspect visuel du décodage progressif.

Il est à noter que ces critères s'appliquent à la fois pour l'original (en pleine résolution) et les représentations intermédiaires des plantes.

La figure 2.9 (à droite) montre comment nous pouvons combiner ces filtres.

Les courbes *modèles*

Le processus précédent nous a permis d'obtenir un ensemble de groupes contenant des représentations normalisées des courbes de Bézier. Nous pouvons maintenant calculer les courbes modèles pour chaque groupe en tant que moyenne des autres courbes.

Expression des détails et des instances

Pour chaque courbe dans un groupe, nous codons maintenant les différences des ses points de contrôle à la courbe modèle (c.f. figure 2.12). Nous appelons ces différences *vecteurs de détail*.

Pour chaque courbe, nous pouvons également définir des paramètres d'instantiation. Ils constituent les exigences minimales qui permettront au décodeur de tirer un cylindre généralisé à partir des modèles fin de rendre une approximation des branches.

L'encodage d'un cylindre généralisé est maintenant défini par cinq entités : la branche modèle, le rayon de modèle, le paramètres d'instantiation, les détails de la branche et les détails du rayon.

A.2.3.3 Codage binaire

Après la transformation de nos courbes de Bézier connectées en un représentation progressive, nous obtenons trois classes de données : *modèles*, *instances* et *détails*. Nous codons maintenant de façon efficace pour construire un ensemble de « paquets de données » interdépendantes, appelés *morceaux binaires*. Quelques informations générales, nécessaires pour le décodeur, seront agglomérées en un bloc de données non classés : l'en-tête.

Pour les principales catégories de données, nous tenons à exprimer ici les paramètres qui doivent être codés pour être en mesure de décoder progressivement une plante.

- **Les modèles :** Tout d'abord, notons que pour les branches, les premier et dernier points de contrôle n'ont pas besoin d'être codés : ce sont toujours (0,0,0) et (0,0,1). Pour faire référence à la fois au modèle de la branche et du rayon lors du décodage, nous avons besoin de définir un *identifiant* de modèle.

- **Les instances :** Pour instancier un modèle de branche sur l'arbre, il nous faut d'abord une référence de sa branche mère, qui est une autre instance. Cela requiert le codage d'un identifiant d'instance. Ensuite, pour « attacher » la courbe à sa mère, nous avons besoin du paramètre u. Enfin, nous avons besoin de référencer puis de transformer les modèle afin qu'ils regagnent leur aspect avant normalisation.

- **Les détails :** Comme pour les modèles les branches, les détails requièrent le codage des vecteurs en 3D pour les branches et en 2D pour les rayons. En outre, l'on doit faire référence à la branche à laquelle appartiennent les détails.

Outre les vecteurs de détails, nous codons simplement les paramètres : les scalaires, les scalaires uniformément répartis dans un intervalle fixe et les entiers (toujours positifs et bornés).

Un avantage du codage différentiel multi-résolution réside en la capacité à quantifier les vecteurs de détail, à l'aide d'un petit nombre de bits, et de choisir un représentant binaire pour chaque valeur en fonction de leur distribution.

Nous avons étudié un méthode de quantification relativement simple, dont l'évaluation de l'entropie nous a montré que l'utilisation d'un codeur entropique serait fort bénéfique. Nous utilisons donc un codeur de Huffman (c.f. [Huf52]) afin d'attribuer un symbole binaire à chaque coordonnée de vecteur détail.

Ainsi, le procédé de codage binaire se termine sur une ensemble de *petits* paquets de données binaires (que nous nommons *morceaux binaires* afin de les distinguer des « paquets », unités de transmission sur un réseau de type IP). Ces morceaux binaires sont interdépendants, leur format précis est décrit dans la figure 2.16 et leurs dépendances dans la figure 2.17.

A.2.3.4 Métrique de qualité

Comme notre but est de décoder progressivement les morceaux binaires, i.e. de fournir un maximum de qualité de rendu pour l'utilisateur, nous avons besoin de ordonnancer nos morceaux binaires. Il n'y a évidemment pas de cycles dans les dépendances exprimées dans la section précédente ; Cela nous donne un premier ordre partiel sur les morceaux binaires : un graphe direct acyclique (appelé DAG, *Direct Acyclic Graph*). Cette ordre « envoie » d'abord les morceaux qui sont décodables, i.e. ceux dont les dépendances ont déjà été envoyées[9].

Cependant, cet order fondé sur les dépendances n'est pas total. Pour ordonner les morceaux binaires « prêts-à-envoyer » nous avons besoin d'une métrique de qualité qui nous assurera que nous accordons la priorité aux morceaux binaires qui maximisent la qualité de rendu de la plante partiellement reconstruite. Par conséquent, nous voulons mesurer la contribution visuelle, ou *importance*, d'un morceau binaire.

[9]Pour l'instant, nous ne considérons pas les pertes de paquets ou réordonnancements.

La contribution visuelle d'un morceau binaire à un modèle rendu dépend de la perception subjective de l'utilisateur. Nous avons ainsi proposé une première métrique facile à calculer et adaptable fondée sur des considérations géométriques. Cette métrique nous permet de formuler un ordre total sur les morceaux binaires quasi-optimal si le décodage se fait dans cet ordre (c.f. figure 2.19) ; nous appelons cet ordonnancement : *FIFO*.

A.2.4 Résultats expérimentaux

Nous avons soumis notre système de compression progressive à plusieurs expérimentations. Nous avons d'abord effectué des expériences sur la normalisation et le regroupement des courbes par rapport à des critères variés. Ensuite, nous avons évalué quantitativement les taux de compression atteints par notre méthode. Les résultats, qui sont fort encourageants, se trouvent dans la section 2.4.

A.2.5 Conclusion

Nous avons proposé une représentation originale progressive des systèmes de ramifications. Cette représentation permet de compresser efficacement des plantes représentées par des cylindres généralisés. Notre méthode fournit un ensemble de morceaux binaires interdépendants, qui seront utilisés dans la prochaine section pour être mis en paquet et être transmis sur des réseaux de type « Best Effort » à l'aide de la métrique de qualité proposée.

A.3 Paquétisation et transmission d'objets 3D

L'étude précédente nous a conduit à un ensemble de morceaux binaires interdépendants représentant progressivement un modèle de plante. Les morceaux binaires sont classés à l'aide d'une importance calculée grâce à une métrique de qualité. Cette ordre fait en sorte que, *si les morceaux sont décodés dans le même ordre*, la qualité de la plante progressivement reconstruite est (presque) maximale pendant le décodage. Dans le cas d'une transmission sans perte et odonnée (e.g. dans un flux TCP), cela est vrai, mais dans le cas général d'une transmission à base de paquets avec possibilité de pertes (e.g. UDP), ce ne l'est point.

Dans cette section, nous étudions la mise en paquets et la transmission de modèles 3D multi-résolution génériques sur réseaux avec pertes, en particulier, nous considérons nos modèles de plantes progressifs et les *progressive meshes* (ou maillages progressifs). Après une brève introduction au contexte de réseaux *best effort* (section A.3.1), nous rationalisons le problème de la mise en paquet (section A.3.2). Ensuite, nous donnons un aperçu des travaux antérieurs liés à la transmission et la paquétisation de contenu 3D (section A.4). Dans la section A.4.1, nous présentons notre modèle analytique et son application à la paquétisation de nos morceaux binaires interdépendants. Et enfin, avant de conclure (section A.4.5),

nous citons les études expérimentales que nous avons effectué sur réseau grande échelle (un WAN : *Wide Area Network*).

A.3.1 Le multimédia, les réseaux… le streaming

En général, il existe deux méthodes principales permettant l'accès aux contenus disponibles à distance via un réseau. La première est le téléchargement d'un fichier, suivi de son utilisation (visualisation, calcul, etc.). Télécharger est la base de l'Internet : par exemple, les pages d'un site Web sont téléchargées par l'intermédiaire du protocole HTTP[10], avant d'être rendues, ou présentées, à l'utilisateur ; les e-mails sont échangés par SMTP[11] par les serveurs sous forme de téléchargement ; de la musique et des films sont massivement échangés chaque jour par téléchargement en Peer-to-Peer (P2P). La deuxième méthode est le *streaming*, ou transmission progressive. Le streaming multimédia consiste à présenter constamment les médias à un utilisateur final pendant leur transmission. Les premiers succès mondiaux de mise en œuvre d'applications de streaming, ont été les « radios internet », par exemple, le service *SHOUTcast* qui, à l'aide du protocole HTTP, a permis pour la diffusion audio sur l'Internet pendant plus de 10 ans. Maintenant les applications streaming vidéo (sur HTTP aussi) sont également courantes. Le streaming multimédia permet d'accéder de manière *plus ou moins* interactive à de très grands contenus, progressivement, i.e. sans attendre leur téléchargement complet.

Dans notre cas, tel que présenté dans le premier chapitre, de scènes 3D, les contenus sont très grands et la visite d'un environnement virtuel est une application très interactive. Le streaming progressif est donc un moyen naturel d'accéder à des objets 3D dans ces applications.

Compte tenu de l'état des réseaux d'aujourd'hui, le streaming multimédia est une tâche difficile. L'Internet est fondé sur un réseau de réseaux *best effort*, il n'y a aucune Qualité de Service (QoS) garantie. L'Internet est un réseau à commutation de paquets, des noeuds du réseau (e.g. les routeurs) peuvent mettre dans une file d'attente et, par conséquent, retarder, un paquet à tout moment, et, si leur file d'attente est pleine, ils peuvent le laisser tomber sans fournir aucune information à l'expéditeur. Cette architecture conduit aux caractéristiques principales suivantes :

- le délai est variable (*gigue*) ;
- la bande passante est variable ;
- il existe des pertes de paquets aléatoires et des désordonnancements (*desequencing*).

Par conséquent il revient aux systèmes d'exploitation et aux applications d'assurer à nouveau la fiabilité et la qualité de service en traitant de ces caractéristiques de bout en bout.

[10] HyperText Transfer Protocol c.f. [FGM+99]
[11] Simple Mail Transfer Protocol c.f. [Kle08]

Avec les années, les conditions se sont évidemment améliorées ; de plus « gros » tuyaux ont été installés ainsi que des routeurs plus « intelligents » (voir par exemple les méthodes « Random Early Detection » [FJ93]). Mais des problèmes demeurent, par exemple, les réseaux sans-fil et/ou mobiles. En outre, les demandes sont de plus en plus exigeantes, par exemple, pour le streaming vidéo, les applications ont besoin de délais de démarrage et de commutation de canaux plus petits.

En ce qui concerne le modèle multi-niveau des réseaux (souvent appelé *OSI* c.f. [Tan02]), nous nous concentrons maintenant sur le niveau « transport ». Pour gérer les exigences des applications distribuées (streaming ou non), la couche transport est dominée par le protocole TCP (le *Transmission Control Protocol*, c.f. [Pos81]) et UDP (*User Datagram Protocol*, c.f. [Pos80]). TCP est le plus souvent utilisé pour télécharger et pour du streaming moins interactif, et UDP est utilisé pour les applications très interactives et temps réel.

TCP est un protocole orienté « flux » assurant fiabilité et ordre et des mécanismes de contrôle de flux et de congestion. TCP fournit des communications « full-duplex » (i.e. dans les deux sens) entre deux pairs connectés. Le contrôle d'erreur est le service qui assure que les éléments de données sont envoyés et reçus dans le même ordre, sans pertes, grâce à un système de retransmission. Les contrôles de congestion et de flux font en sorte que le réseau et le récepteur respectivement ne sont pas envahis par une quantité de paquets qu'ils ne peuvent pas gérer. TCP utilise principalement un système de contrôle de congestion AIMD (*Additive Increase Multiplicative Decrease*) fondée sur la détection de perte : chaque unité de temps, l'émetteur augmente sa fenêtre d'envoi d'une unité, mais si une perte est détectée, il divise la fenêtre d'envoi par deux. Notez qu'une perte de paquet est interprétée comme une congestion dans le chemin sur le réseau, cela n'est guère toujours vrai, mais cette hypothèse conservatrice permet de fournir une méthode très fiable.

UDP est un protocole orienté « datagramme » sans connexion. Il fournit le minimum de service aux applications, en fait, les seuls services fournis lors de l'envoi de paquets sont la distribution des datagrammes entre les applications (*le numéro de port UDP*), et une détection d'erreurs de transmission spartiate (sur la base d'une somme de contrôle sur un champ de 16-bit). UDP est conçu pour des applications qui connaissent mieux ce qu'il faut faire (ou pas) en cas de perte de paquets, et pour des applications dont les concepteurs ont pensé que l'établissement d'une connexion TCP serait beaucoup trop lourde par rapport aux besoins de transmission de données (e.g. les service DNS, *Domain Name System*).

UDP peut être utilisé par exemple pour le streaming vidéo, mais lorsque l'application vise les utilisateurs de l'Internet, les concepteurs ont souvent à se replier sur TCP. Le problème est que UDP n'est pas un grand ami des sous-réseaux cachés par un mécanisme NAT (*Network Address Translation*), et en fait, la plupart des réseaux domestiques courants sont des NAT (d'ailleurs, souvent avec un seul ordinateur). UDP est plutôt utilisé lorsque le fournisseur de services est propriétaire de l'ensemble du chemin (e.g. les réseaux d'opérateurs mobiles).

Pour les deux protocoles TCP et UDP, les unités de données applicatives (ADUs) ont besoin d'être fragmentées et/ou emballées dans des unités de données réseau (NDUs). Dans le monde TCP, les NDUs sont appelés *segments*, chez UDP, ils sont appelés *datagrammes*. Le processus de fragmentation et d'emballage, ainsi que l'ordonnancement des paquets, est généralement appelé *paquétisation*. Le système d'exploitation peut prendre soin de la mise en paquet dans le cas de TCP (cela est en fait le comportement par défaut). Mais pour UDP et/ou des applications qui nécessitent une des réglages *fins* à des fins de performance, la paquétisation doit être traitée au niveau de l'application, i.e. en étant conscient des caractéristiques des données transmises.

Cela est en particulier le cas pour le streaming des objets 3D. Cette application a ses propres exigences (c.f. le premier chapitre), et les modèles 3D ont une structure de dépendance spécifique (c.f. section A.3.2.1). La suite détaille le problème de la mise en paquet pour le streaming des objets 3D.

A.3.2 Le problème de la mise en paquets

Nous considérons désormais la transmission à base de paquets de morceaux binaires représentant un modèle 3D progressif. En règle générale, car nous considérons les morceaux binaires plus petits que les paquets, nous avons à mettre en paquet un certain nombre de morceaux binaires (primitive de la modélisation des données d'un point de vue applicatif, ou ADU) à l'intérieur d'un paquet (unité de transfert du réseau, ou NDU). Comme dans la section A.2.3.4, ces morceaux peuvent être totalement ordonnés grâce à une métrique de qualité. Dans cette section, nous expliquerons plus précisément les questions que nous allons aborder.

A.3.2.1 Caractéristiques de nos données

Dans ce chapitre, nous considérons que les données en entrée sont un ensemble de morceaux binaires interdépendants qui peuvent être le résultat de notre système de compression d'arborescences de cylindres généralisés (c.f. section A.2.3.3), ou d'une autre méthode de codage progressif de données 3D, comme les *Progressive Meshes* (présentés par Hughes Hoppe dans [Hop96]). Ainsi, la plupart des codages multi-résolution de maillages triangulaires [AD01, AG05, Tau99, DG00b], à base de points [Pau03, RL00, KB04, FACOS03] ou de représentations hybrides [CN01] conduisent à l'interdépendance des données. Les dépendances entre les éléments primitifs de données ont la même « macro-forme », elles peuvent être représentées, de manière générique, par un DAG (Graphe Direct Acyclique). La figure 3.3 montre une structure de dépendance : si A et B sont des morceaux binaires (ou des paquets), une arrête dans le graphe entre A et B signifie que B dépend de A, en d'autres termes, que pour décoder B nous avons besoin d'avoir décodé A. La caractéristique principale de ces dépendances de décodage est que les résolutions les plus grossières sont importantes pour décoder les plus fines.

A.3.3 Pourquoi donc retransmettons-nous des paquets?

Dans le cas idéal où les données sont reçues l'ordre dans lequel elles ont été envoyées, l'ordonnancement par décodabilité (les dépendances) aidé par une métrique de qualité (comme dans la section A.2.3.4) peut être considéré comme quasiment optimal. Si les éléments de données (les morceaux binaires) sont emballés ensemble pour remplir des paquets de taille prédéfinie l'on une première stratégie de paquétisation appelée FIFO (*First In First Out*).

Ce cas est observé, par exemple lorsque les données sont transmises au moyen d'un flux TCP. En effet, le *Transmission Control Protocol* assure que les éléments de données sont envoyés et reçus dans le même ordre et sans pertes.

Mais, comme présenté dans la section A.3.1, les réseaux IP génériques ne fournissent que des transmissions non-fiables. Les protocoles qui assurent des connexions ordonnées et sans perte doivent retransmettre les données potentiellement perdues, et, au cours de la retransmission d'un paquet, les paquets qui suivent le paquet retransmis doivent être mis en tampon pour les délivrer à l'application dans l'ordre original. Par exemple, sur le côté expéditeur, TCP code l'ordre de segment comme un numéro de séquence d'octets dans l'en-tête du protocole. Côté client, si un segment est perdu, la pile TCP met en tampon les segments qui ont un plus grand numéro de séquence, jusqu'à ce que l'émetteur détecte la perte de paquet (grâce à l'absence d'accusé de réception après un délai) et que le segment incriminé soit retransmis et effectivement reçu.

Ce phénomène de mise en tampon nous amène à une observation simple : avec un protocole comme TCP qui assure des transmissions ordonnées et sans perte, lorsque se produit une perte de paquets, un certain nombre de paquets sont potentiellement arrivés sur le poste client, mais sont conservés par la pile réseau du système d'exploitation et donc de ne sont pas livrés à l'application. En d'autres termes, certaines données sont disponibles sur l'ordinateur du client, mais l'application ne peut pas les utiliser pour améliorer la qualité visuelle du rendu des objets.

C'est pour cette raison, que nous concentrons nos études les protocoles de transmission à base de datagrammes, i.e. sans ordre intégré. L'on peut ainsi tirer profit de tous les paquets arrivant afin d'améliorer la qualité du rendu partiel du modèle.

Les protocoles basés datagramme couramment utilisés apportent plus de flexibilité, mais également ne masquent pas les pertes de paquets. Il existe plusieurs façons de traiter la perte de paquets (c.f. section 3.3). Nous choisissons la *retransmission* : lorsqu'une perte est détectée par l'expéditeur, le paquet incriminé est retransmis. Les pertes dans le cas de contenu 3D ne sont guère souvent récupérables et peuvent causer des défauts de rendu durables. C'est précisément la principale différence entre la perte de paquets pour la 3D et pour la vidéo (en streaming) : la *persistance* (ou *durabilité*) des artefacts visuels induits. En effet, une perte dans un flux vidéo, même pour une image *clé*, cause des conséquences visuelles seulement pour, au plus, quelques secondes. Pour un modèle en 3D, un maillage progressif ou une plante, un *trou* dans la géométrie peut rester visible pendant toute la visualisa-

tion de l'objet et peut empêcher un grand nombre de données d'être décodé. Cette différence est visible dans les graphes de dépendance de ces éléments de données (figure 3.1).

Notre étude est donc basée sur les protocoles datagramme avec un mécanisme de retransmission afin d'assurer la fiabilité. Par exemple, nous avons utilisé dans nos expériences UDP+R (*User Datagram Protocol avec Retransmission*) ou DCCP+R (*Datagram Congestion Control Protocol avec Retransmission* c.f. [KHF06]) lorsque l'aide d'un système de contrôle de congestion est nécessaire.

Ces choix donnent un cadre à notre étude sur la paquétisation mais nous avons pour objectif de ne pas fermer la porte aux codes FEC (*Forward Error Correction*) ou la multi-résolution fondée sur la prédiction géométrique, topologique ou numérique des données (c.f. section A.4.5).

A.3.3.1 Un problème de dépendances

Dans ce contexte de transmissions basées datagramme et retransmission, nous nous concentrons sur l'ordre de réception des paquets. Du côté de l'expéditeur, les paquets sont envoyés dans un certain ordre, mais cet ordre peut être différent sur le client. Deux raisons principales peuvent conduire à un désordonnancement des paquets.

- Deux paquets peuvent prendre différentes *routes* dans le réseau, ce cas est très rare, et conduit généralement à de très petits retards ; il n'est donc pas considéré dans notre étude.

- Le mécanisme de retransmission peut, en cas de perte, retarder des paquets : pendant le temps nécessaire pour détecter une perte de paquets et de les retransmettre, de nombreux paquets suivants ont le temps d'atteindre le récepteur.

Il y a deux mécanismes de retransmission principaux : le *TCP-like* et les *NACK-based*. Dans le protocole TCP, le récepteur envoie un accusé de réception (acquittement ou *ACK*) pour chaque segment reçu (les ACKs peuvent être emballés ensemble dans une même paquet pour accroître la performance). Quand un paquet est envoyé, l'expéditeur lance une minuterie, si le paquet n'est pas acquitté par le récepteur avant l'expiration de la minuterie expéditeur considère qu'il a détecté une perte, et prévoit donc la retransmission du paquet incriminé. D'autre part, la retransmission peut être fondée sur la détection de perte par le récepteur qui envoie donc un demande de retransmission. Le récepteur peut détecter un saut dans les numéros de séquence et/ou en utilisant aussi un minuteur. Dans ce cas, le récepteur envoie une requête de retransmission, appelée NACK i.e. non-acquittement. Les deux méthodes de retransmission conduisent à un retard évident du paquet perdu, et donc de désordonnancement des paquets sur le client (c.f. figure 3.2).

Le désordonnancement des paquets induits par les retransmissions deviennent problématiques lorsque l'on considère les dépendances entre paquets. Par exemple,

si les inter-dépendances des paquets sont modélisées comme dans la figure 3.3, pendant la retransmission éventuelle du paquet 7, les paquets 14 et 15 peuvent être arrivés sur le client mais ne pas être décodables à cause des dépendances.

Par conséquent, le problème de dépendance, auquel nous devons faire face est : **Comme une perte/retransmission peut retarder le décodage de données déjà arrivées, nous voulons optimiser, à tout moment, le quantité de paquets décodables parmi les paquets arrivées.** Et d'améliorer ainsi la qualité du modèle partiellement décodé dans un environnement réseau sujet aux pertes.

A.3.4 Dépendance et qualité

Le problème de dépendance déjà exprimé ne devrait pas être considéré seul. Afin d'optimiser la qualité de l'expérience du client/spectateur, il nous faut prendre en compte à la fois les dépendances et de l'importance des morceaux binaires.

Dans la section A.3.3, nous avons présenté une première stratégie de mise en paquet appelée FIFO, cette stratégie tente de maximiser la qualité de partie décodée d'un modèle, mais prend naïvement la dépendance entre les paquets en compte. D'autre part, une stratégie de paquétisation comme celle proposée dans [GO05] (c.f. section 3.3), qui réduit la dépendance entre les paquets, vise à remédier problème de dépendances, mais étant donné que la qualité n'a pas été considérée, un algorithme peut emballer et envoyer des morceaux binaires moins importants en premier, et, par conséquent, être moins efficace à l'amélioration de la qualité sur le client, surtout quand il n'y a pas de pertes.

Ainsi, notre objectif dans cette section est de fournir une réponse au problème de l'optimisation de la qualité du modèle décodé en prenant en compte les pertes dues à l'environnement réseau.

A.4 Paquétisation et transmission d'objets 3D

Dans la section 3.3, nous proposons un état de l'art des travaux en cours sur la mise en paquet et la transmission efficace d'objets 3D, i.e. l'aspect bas-niveau de la diffusion des modèles 3D, le streaming générique des scènes 3D étant étudié dans le chapitre suivant.

La plupart des travaux porte sur la transmission de maillages 3D, principalement des représentations multi-résolution de maillages triangulaires. Il existe trois grandes classes de travaux :

- la compression et le codage tolérants aux pertes (section 3.3.1) ;
- le controle d'error (section 3.3.2) ;
- et la mise en paquets adaptée (section 3.3.3).

A.4.1 Un modèle analytique pour le streaming 3D

Les études existantes se concentrent soit sur les dépendances soit sur l'importance des données. Ces deux facteurs influent sur la qualité du décodage des objets 3D. Aucun, cependant, ne s'est penché sur ces deux facteurs à la fois et n'a caractérisé leurs effets sur la qualité. Nous visons à atteindre cet objectif en proposant un modèle analytique.

Nous proposons un modèle analytique pour la diffusion progressive des données 3D qui mène à une stratégie améliorée de mise en paquet nommée *Greedy*. Ceci est un travail commun avec l'université de Singapour (NUS) qui a été initié par Wei Cheng et Wei Tsang Ooi[12]. Des détails sur ce modèle (les preuves, et certaines expériences) peuvent être trouvés dans [COM+07] et dans [COM+09].

A.4.2 Le modèle

Notre modèle analytique considère l'envoi de paquets par un expéditeur à une vitesse moyenne d'un paquet par unité de temps. Nous considérons un protocole utilisant une technique de retransmission, les deux type des protocoles *NACK- et ACK-based* sont compatibles.

L'idée est d'abord d'exprimer l'espérance des dates d'envoi et de réception en fonction du taux des pertes et du temps nécessaire à la détection d'une perte, ce dernier dépend du RTT, *Round Trip Time*. En effet, notons que la date d'envoi effective d'un paquet est toujours retardée par les retransmissions des paquets précédents.

Ensuite l'on introduit les dépendances entre les paquets afin de calculer leur date de décodage.

Notre modèle analytique est utile à plusieurs égards ; les équations peuvent nous aider à comprendre les effets des dépendances lors de la transmission d'un objet 3D progressif sur un réseau avec pertes. Nous pouvons également calculer l'espérance de qualité de l'object décodé, conduisant à une alternative plus rapide que la simulation comme moyen d'évaluer les effets des conditions du réseau sur la transmission progressive (c.f. [COM+07] et [COM+09]).

A.4.3 Améliorer la transmission des objets 3D

Pour donner une réponse au problème présenté dans A.3.4, i.e. réduire les dépendances entre paquets tout en gardant l'objectif de maximiser la qualité du décodage partiel d'un modèle 3D, nous proposons une stratégie de paquétisation qui tient compte à la fois de l'importance des morceaux binaires et des dépendances entre ceux-ci. L'idée principale de la méthode est, sur la base du modèle analytique, de donner une évaluation quantitative du compromis à faire entre la maximisation de la qualité du décodage et la minimisation de la dépendance entre les paquets.

[12] c.f. nemesys.comp.nus.edu.sg/projects/3dstream

Pour cela, nous définissons d'abord ce que l'on entend exactement par « qualité du modèle partiellement décodé » : puisque nous voulons améliorer la qualité du rendu de modèle à tout moment, et en particulier au début de la session de streaming, nous évaluons la qualité intermédiaire, sur une période de temps plutôt qu'à une certain instant, nous pouvons la définir comme une intégrale sur toute la transmission courante de l'importance des morceaux binaires qui ont été décodés.

Nous pouvons ensuite définir notre algorithme de mise en paquet : si nous considerons un morceau binaire c, nous devons décider si nous devons emballer c dans le paquet actuel ou si nous pouvons le laisser pour le paquet suivant.

Tout d'abord, nous constatons que s'il existe un parent de c qui n'a pas été emballé, alors nous ne devrions pas avoir emballé c (si un parent de c arrive plus tard que c, c ne peut pas être décodé de toute façon). Ainsi, on ne considère que les nœuds dont les parents ont tous été mis en paquet.

Ensuite grâce à l'équation 3.5, nous pouvons évaluer le compromis entre l'importance de c et ses dépendances, en d'autres termes la pénalité imposée par le fait de ne pas mettre c dans le paquet courant. Par conséquent nous pouvons choisir parmi tous les morceaux binaires dont les dépendances autorisent la mise en paquet, celui dont la pénalité est la plus faible.

A.4.4 Évaluation des performances

Notre objectif fut de fournir des résultats expérimentaux sur la transmission progressive des modèles 3D sur un « vrai » réseau du monde réel (i.e. l'Internet). Nous avons réalisé des expériences avec des buts différents :

1. valider les preuves et les hypothèses du modèle analytique ;
2. comparer l'efficacité des stratégies de paquétisation FIFO et Greedy ;
3. expérimenter la transmission avec des morceaux binaires résultant de la compression progressive de plantes présentée dans le chapitre 2.

Nous présentons dans la section 3.5 les résultats pour les deux derniers objectifs concernant l'algorithme Greedy. La validation du modèle analytique peut être trouvée dans [COM+07] et dans [COM+09].

A.4.5 Conclusion et perspectives

Dans ce chapitre, nous avons montré comment on peut transmettre des données interdépendantes résultant de la compression progressive de modèles 3D sur des réseaux *best effort* avec pertes. Les expériences montrent que la stratégie de mise en paquet Greedy surpasse FIFO, lorsque le taux de perte est élevé et au cours des premiers quelques multiples du RTT. Dans les applications interactives de NVE, ces premières secondes sont importantes dans le sens où un utilisateur, lors de la navigation, passera à proximité de nombreux objets 3D qui ne seront visibles que pendant de courtes périodes de temps.

A.5 Modélisation et streaming de scènes 3D naturelles

Précédemment, nous avons abordé les problèmes relatifs à la diffusion des objets 3D. D'abord avec la représentation progressive d'une modélisation particulière, et ensuite par la mise en paquet d'objets 3D multi-résolution génériques. Dans cette section, nous prenons un point de vue plus élevé, et nous étudions la diffusion de *scènes 3D* composées de divers objets 3D. Nous conservons l'accent mis sur les scènes naturelles, et nous utilisons des techniques d'adaptation visant à améliorer un système de streaming.

La section A.5.1 détails ce que *sont* les scènes 3D. Puis, dans la section A.5.2, nous discutons de l'état de l'art sur la distribution de scènes 3D. Une première technique d'adaptation est proposée dans la section A.5.3.

A.5.1 Des scènes 3D naturelles

D'un point de vue grossier, une scène 3D est tout simplement une collection d'objets représentant des éléments d'un monde virtuel. Ces objets sont souvent appelés « entités ». Les entités peuvent représenter des objets solides, mais elles peuvent également être des lumières (i.e. paramètres représentant l'illumination de la scène), des paramètres de brouillard ou de vent... Même les objets géométriques peuvent être modélisée en utilisant des techniques totalement différentes, par exemple, une scène en peut consister en un terrain représenté par une carte de hauteurs, les bâtiments modélisés en utilisant des maillages rectangulaires, les arbres modélisés comme L-Systems, etc.

Les entités peuvent aussi n'être que des transformations ou autres paramétrages d'objets plus génériques. Ces entités sont alors dans ce cas « instances », elles instancient des « modèles ». Par exemple, nous pouvons avoir un modèle de maison générique, décrivant la géométrie, la couleur, etc d'un bâtiment, d'une part. Et d'autre part, beaucoup des instances *légères* de la maison, qui ne sont que la position réelle de la maison et un peu de paramètres de déformation (afin de ne pas avoir partout exactement la même maison dans la scène). Ce mécanisme est appelé « instantiation », il permet de réutiliser le contenu, et, ainsi, d'alléger la quantité de données nécessaires pour décrire la scène.

Un autre mécanisme courant est le fait de permettre aux entités d'être elles-mêmes des collections d'entités. Par exemple, dans une scène naturelle, une forêt peut être modélisée comme une entité qui contient un certain nombre d'arbres. Cette idée, qui permet de structurer la scène en tant que arborescence, facilite les traitements communs sur les familles d'objets. Il peut aussi être vu, dans une modélisation objet, comme un patron de conception « Composite » (c.f. [GHJV95]).

Dans notre travail, nous nous concentrons sur des scènes naturelles, nous devons donc prendre en considération leurs spécificités.

- Tout d'abord, les scènes naturelles sont généralement grandes ; le plus souvent, elles sont géographiquement vastes, mais la préoccupation d'un système de streaming est la taille des données, et dans ce cas, même un jardin

japonais peut contenir des modèles très complexes et, ainsi, de lourdes structures de données.

- La seconde spécificité est la répartition spatiale non-uniforme des objets dans un paysage naturel. Comme dans le monde réel, la densité de objets dans un environnement naturel est très hétérogène, il peut y avoir clairières, des parcelles de forêt dense, etc.

- Enfin, par rapport à des scènes comme les zones urbaines ou les environnements intérieurs, les scènes naturelles souffrent de très peu d'occlusion entre les entités. Ce manque d'occlusion induit le plus souvent que nombreuses entités sont visibles en même temps.

Nous nous efforçons de répondre à ces spécificités, de toute évidence d'un point de vue streaming, dans la section A.5.3.

A.5.2 État de l'art

Nous avons classé les systèmes de streaming de scnènes 3D en trois catégories: les client-serveur classiques, le peer-to-peer (P2P), et les systèmes « basés vidéo ».

- **Les systèmes client/serveur :** Un système de streaming 3D avec un modèle client-serveur, consiste en un hôte (le serveur) contenant de la scène 3D et les utilisateurs (les clients) accédant à la scène interactivement. L'on peut citer par exemple le projet ARTE (c.f. [Mar00]) fondé entre autres sur des travaux en compression des maillages (c.f. [Tau99]). Des travaux ont été effectués dans le domaine de l'adaptation au point de vue (e.g. [COZ98, KLK04, CO08]), de même que des études se plaçant au niveau « scène » (e.g. [TL01, RGCB07]).

- **Les systèmes pair-à-pair :** Le *Peer-To-Peer* est une cible prometteuse pour la diffusion de scènes 3D. Un premier système fut proposé dans [CBR06] pour les environnements urbains 2.5D. D'autres projets on suivit avec un accent plus fort sur les modèles 3D : *HyperVerse* (c.f. [BHS+08]), ASCEND (dans [Hu06, SHJ08]).

- **Les systèmes fondés sur du streaming vidéo :** Au lieu de transmettre du contenu 3D et laisser le client rendre la géométrie, [CBPEZ04] et [NCO03] proposent de rendre la scène sur le serveur puis de diffuser de la vidéo. Étant donné le point de vue du client (envoyé comme une requête), le serveur rend une vidéo MPEG-4 la transmet au client. Cette approche semble peu-échelonnable de part sa charge sur la mémoire et le CPU du serveur pour chaque client. Mais quand l'on cible des dispositifs mobiles (PDA ou téléphones) qui ont des capacités fort limitées de rendu 3D, cette technique peut parfois être la seule solution.

A.5.3 L'adaptation du côté du serveur

Comme expliqué plus haut, d'un point de vue « gestion de scène », les objets peuvent être des entités de tout type (e.g. points, maillages, plantes procédurales...). Pour représenter des objets géométriques au niveau de la scène et de fournir une interface uniforme, une méthode courante consiste à utiliser leur boîte englobante alignée sur les axes (AABB), la valeur minimale et maximale de chaque coordonnée de l'objet (d'autres volumes sont parfois utilisés, voir par exemple [BCG+96]).

Par ailleurs, tout en travaillant avec de grandes scènes naturelles, certaines caractéristiques spécifiques nous sont apparues :

- Les scènes ont, à un niveau de détail grossier, une topologie en deux dimensions (ou *surfacique*) : la plupart des scènes (et surtout naturelles) sont composés d'un grand terrain et de beaucoup de petits objets posés sur celui-ci.

- La distribution des objets dans la scène est très irrégulière. Une scène consiste généralement en des groupes d'objets très denses (e.g. une forêt est un groupe d'arbres) et de grandes régions d'espace vide.

Nous visons à développer une gestion efficace d'une grande scène naturelle sur le serveur : pour la distribution de contenu adapté aux différents clients, le serveur ne doit envoyer que le minimum d'objets visibles selon le point de vue du client. Le volume visible est appelé *frustum* (c.f. figure 4.1). Nous avons besoin d'une structure de données afin de gérer, au niveau de la scène, un grand nombre d'AABB représentant les objets.

Notre objectif est d'optimiser en temps réel les requêtes sur le serveur : les rendre précises et efficaces. Il s'agit d'améliorer la traversée de la structure de données pour un « abattage » des objets non-visibles (ou « frustum culling », en d'autres termes, sélection des objets visibles). La création de la structure de données peut être coûteuse, et faite hors-ligne. La plupart des structures de données pour les scènes ont été motivées par des considérations de rendu où la quantité d'objets visibles est moins critique que pour le streaming, où la bande passante du réseau est un goulot d'étranglement.

Nous avons donc défini, et validé par des expérimentations, une structure de données qui permet de sélectionner un sous-ensemble d'objets visibles de la scène (les candidats immédiats au streaming). Nous nous proposons de combiner les avantages des « Bounding Volume Hierarchies » à l'adaptation à la topologie de scènes réelles. Nous construisons les arbres binaires d'AABB en optimisant de la *coupe*, i.e. la séparation d'une boîte englobante en deux, afin de la rendre aussi *discriminante* que faire ce peut. Au final, notre structure de données optimise la vitesse des requêtes de point de vue et se montre plus précise que ses concurrentes.

A.6 Aspects pratiques et leçons retenues

Dans cette thèse nous avons abordé les divers problèmes avec une approche « systèmes » et « ingénierie ». Toutes les idées fournies ont été testées et validées dans des applications ciblant le monde réel. Contrairement à de nombreux projets industriels, un travail doctorat en général bénéficie d'un départ libre. Cela nous a donné la possibilité de choisir avec soin les outils que nous avons estimé qu'ils conduiraient à un développement de logiciels simple, sûr et efficace, et donc de valider nos contributions théoriques.

Dans ce chapitre, nous traitons les aspects pratiques du travail que nous avons accompli. Dans la section A.6.1, nous discutons sur les leçons que nous avons tirées de l'expérience et les outils que nous avons utilisé et évalué tout au long de la thèse. Ensuite, nous donnons une récapitulation des systèmes logiciels qui ont été mis en œuvre (section A.6.2).

A.6.1 Outils et Leçons

Avant de s'aventurer sur son champ de bataille, un étudiant en doctorat de *choisir ses armes*. Nous présentons ici les outils et les idées de conception qui ont été appris, utilisés et évalués au cours de ces études.

A.6.1.1 Systèmes d'exploitation et Réseaux

Nous avons d'abord choisi un terrain d'exploitation commun pour l'expérimentation. La seule plate-forme qui s'est avérée assez mûre pour gérer les applications multimédia en réseau est *UNIX*. *UNIX* doit être considéré ici comme l'ensemble des exigences de la norme en laquelle il consiste, et non pas, le système d'exploitation aussi appelé *UNIX*. Il peut également être libellé sous la forme « POSIX.1, Core Services » (i.e. IEEE Std 1003.1-1988).

Alors que nous utilisions le système compatible UNIX le plus largement disponible, i.e. GNU/Linux, nous avons eu, au cours de cette thèse, l'occasion de nous écarter de notre exigence de portabilité : le protocole DCCP (c.f. section 3.5), et le rendu 3D.

De manière générale, les systèmes actuellement les plus utilisés (GNU/Linux, MS Windows et MacOSX) se sont révélés ni fiables, ni performants, ni sécurisés. Les systèmes d'exploitation devront évoluer beaucoup pour un avenir meilleur. Toutefois, d'un point de vue « portabilité », les futurs systèmes d'exploitation seront certainement obligés de mettre en œuvre une certaine compatibilité POSIX.

A.6.1.2 Le langage de programmation

Comme disait Andrew Tanenbaum : *« Ajouter du code, ajoute des bugs »* [Tan01]. Les langages de programmation (et la plate-forme de développement) sont surement le plus important choix d'un projet de logiciel. Nous avons choisi le langage Objective Caml pour notre plate-forme de développement.

Objective Caml est la principale implémentation du langage Caml, développé par Xavier Leroy et al. depuis 1985[13]. OCaml unifie programmation fonctionnelle, impérative, et objet dans le cadre d'un système de types de type ML. Il propose notamment un très fort typage statique et inféré (Hindley-Milner), des contrôles de depassement, et un ramasse-mietes aliés à une excellente efficacité [Ler90, Ler00, Gar00].

Caml permet aux développeurs de mettre en œuvre des solutions à des problèmes très complexes (c.f. [MW08]) en très peu de lignes de code, très efficacement, et surtout, en assurant beaucoup de propriétés de sureté dès la compilation :

- Les contrôles de dépassement automatiques et le typage statique fort assurent qu'il n'y a pas de d'accès incorrect à la mémoire, ce qui signifie qu'il ne peut pas y avoir d'incident de segmentation, ou d'erreur de bus. De plus, certaines failles de sécurité qui peuplent abondamment les programmes C et C++ sont nativement évitées : il ne peut pas y avoir de « Buffer Overflow » (c.f. [AO96, MdR99]).

- Il n'y pas de « pointeur NULL » a priori, ainsi les exceptions *NullPointerException* courantes en Java ou Python sont le plus souvent évitées.

- Typage statique, signifie aussi *non-dynamique*, les exceptions classiques de programmes pratiquant le *duck-typing* ne peuvent pas arriver[14].

- Il n'y pas de surcharge des opérateurs, ni de transtypage implicite. Cela enlève les bogues fort difficiles à trouver du type : "`2 / 3 == 0`".

- D'autres caractéristiques ajoutent encore plus de sureté, par exemple, le typage des fonctions de type *printf* évite les failles connues sous l'appellation "printf format injection" (c.f. [Scu01]).

L'implémentation d'Objective Caml fournit un boucle d'interaction (un shell), conjointement avec les compilateurs bytecode et natif (la machine virtuelle est très légère et portable, le code natif s'est avéré très efficace), un générateur de documentation, un analyseur syntaxique extensible nommé Camlp4, un débogueur... et entre autres, beaucoup de librairies.

Pour conclure sur OCaml, après quelques années d'utilisation intensive, et des essais de beaucoup de plates-formes concurrentes, nous pouvons faitre état que Caml n'est pas *Le* language définitif, et il n'est pas parfait, mais il est *de loin* le meilleur compromis que nous avons croisé. Par exemple, Haskell peut être considéré comme la une prochaine étape vers la programmation sûre, mais, pour l'instant, le manque de bibliothèques, et l'imprévisibilité des performances sont aussi des facteurs limitants pour un usage général.

[13] c.f. caml.inria.fr

[14] e.g. en Python l'exception classique :

```
AttributeError: 'int' object has no attribute 'to_int'
```

Nous pensons que les méthodes de programmation telles que le typage statique fort, le contrôle des dépassements, la collecte automatique de la mémoire, le *message-passing*, guidés par une saine paranoïa sur la sûreté et la sécurité, sont le chemin à suivre pour résoudre la plupart des problèmes. Nous devons préciser que nous ne pensons pas que « l'avenir devrait être comme ça », nous pensons que « les vingt dernières années auraient dû être comme ça ».

Nous allons terminer cette section par un dernier conseil empirique sur les langages de programmation :

Ne jugez pas un langage de programmation en vous demandant ce qu'il pourrait permettre à un programmeur intelligent et habile d'accomplir. Au lieu de cela, demandez-vous ce que le langage peut empêcher un programmeur stupide et paresseux de faire. Parce que les programmeurs sont humains, donc… essentiellement… stupides et paresseux.

A.6.1.3 Écrire et présenter

Le deuxième ensemble d'outils utilisés, pour un étudiant en doctorat, est celui qui lui permet d'écrire des articles et de les présenter : les outils de composition de documents.

Les deux outils principalement utilisés ont été LaTeX et Inkscape. Tous deux fort décevants, mais irremplaçables, au vu de la concurrence, pourtant testée (ConTeXt, Lout…).

A.6.1.4 Le travail collaboratif

Nous avons effectué beaucoup de travail collaboratif dans le monde entier : avec et depuis Singapour, Montpellier et au cours de conférences. Pour gérer efficacement ce travail en équipe distribué, des outils de collaboration ont été utilisés. Le premier outil est le *Wiki*, une simple application web de gestion et de rendu de fichiers texte très pratique pour la conception rapide et la discussion. Le deuxième instrument sont les systèmes de contrôle de version. Nous avons mis en place des dépôts pour le code, les articles et les présentations, cela a été très fructueux. La solution retenue fut *Subversion*, cela a été un succès.

A.6.2 Logiciels

Trois principales plates-formes logicielles ont été développées, *LibGenCyl*, *OMAN* et *Wadis*, ils correspondent grossièrement aux chapitres 2, 3 et 4.

A.7 Conclusion

Dans cette thèse, en prenant en compte les particularités du contenu que nous ciblions, les grandes scènes naturelles, nous avons proposé des optimisations sur certains points d'*adaptation* :

- la transformation du contenu afin de l'adapter aux conditions du streaming (chapitre 2) ;
- l'adaptation des systèmes de transmission utilisés pour rendre les objets disponibles sur le réseau (chapitre 3) ;
- l'amélioration de la mise en œuvre sur le serveur de l'adaptation au point de vue et de la préparation du terrain pour les travaux futurs sur le déploiement et l'évolutivité des systèmes de streaming (chapitre 4).

Appendix B

Cast (in order of appearance)

Sebastien Mondet
Affiliation: University of Toulouse
Web Page: sebmdt.googlepages.com

Mathias Paulin
Affiliation: University of Toulouse
Web Page: www.irit.fr/~Mathias.Paulin

Géraldine Morin
Affiliation: University of Toulouse
Web Page: morin.perso.enseeiht.fr

Romulus Grigoras
Affiliation: University of Toulouse
Web Page: grigoras.perso.enseeiht.fr

Wei Tsang Ooi
Affiliation: National University of Singapore
Web Page: www.comp.nus.edu.sg/~ooiwt/

Stefanie Hahmann
Affiliation: Grenoble Institute of Technology
Web Page: ljk.imag.fr/membres/Stefanie.Hahmann/

Eckehard Steinbach
Affiliation: Munich University of Technology
Web Page: www.lmt.ei.tum.de/team/steinb/

Wei Cheng
Affiliation: National University of Singapore

Web Page: www.comp.nus.edu.sg/~chengwe2

Frederic Boudon
Affiliation: CIRAD - Montpellier
Web Page: www-sop.inria.fr/virtualplants/wiki/doku.php?id=team:boudon

Christophe Godin
Affiliation: CIRAD - Montpellier
Web Page: www-sop.inria.fr/virtualplants/wiki/doku.php?id=team:godin

Appendix C

Acknowledgements

First of all, I owe my deepest gratitude to my Ph.D. supervisors, Géraldine Morin, Romulus Grigoras, and Wei Tsang Ooi, this thesis would have gone nowhere without them. Also, I am grateful to Mathias Paulin, for the support and for the honest and valuable advices, and the VORTEX team for welcoming me, and helping me during the whole thesis.

I want to express my gratitude to Stefanie Hahmann and Eckehard Steinbach, for their time and effort on the review of the present thesis.

I am also grateful to Frédéric Boudon, Christophe Godin and the CIRAD of Montpellier, for the interesting discussions, the help and the plant models.

I would like to thank Cheng Wei for the enriching time we had working together, all the students of multimedia systems lab of the NUS, and Nicolas, for the great time spent in Singapore.

I will never thank enough my mother, my brother, and my dear friends and colleagues, Fred, Maitena, Marie, Manu, Ion, Aurore, Cédric, Cécile, Benoît, Caroline, Ludo, Mélanie, Matthias, Béa, Anca, P.Y., Elena, Sandrine, Pascaline, Viorica, Pauline, Omar, Benoît, Jerôme, Christophe, Stéphane, Jean-Charles, Philippe, Cezar, Marco ...

I would like also to thank Elena, Anca, Andra, Sergiu and Sabin, the interns I had the pleasure and the honor to co-advise.

Sylvie, Sabyne, Sylvie, Marie-Blandine and all the staff of the IRIT and the ENSEEIHT, deserve very special thanks; without them, nothing could happen, even the buildings would collapse.

I am grateful also to the musicians I had the honor to play with, and to the

T.E.A.M. Teakwondo club, for helping me preserve my mental health during all these crazy years.

Thanks also, to the whole free software community for providing tools to do the jobs, for always improving them, and making them openly accessible.

Thanks, and apologies to any person I forgot and who deserves thanks.

Finally, I would like to thank all my enemies because they told me my flaws, and all those who hurt me because they made me stronger.

Bibliography

[AD01] Pierre Alliez and Mathieu Desbrun. Progressive Compression for Lossless Transmission of Triangle Meshes. In *SIGGRAPH 2001 Conference Proceedings*, pages 198–205, 2001.

[AG05] Pierre Alliez and Craig Gotsman. *Advances in Multiresolution for Geometric Modelling.*, chapter Recent Advances in Compression of 3D Meshes, pages 3–26. N.A. Dodgson and M.S. Floater and M.A. Sabin. Springer-Verlag, 2005.

[AM00] Ulf Assarsson and Tomas Möller. Optimized view frustum culling algorithms for bounding boxes. *J. Graph. Tools*, 5(1):9–22, 2000.

[AO96] Aleph-One. Smashing The Stack For Fun And Profit. *Phrack*, 7(49), 1996.

[ARA03] Ghassan Al-Regib and Yucel Altunbasak. 3TP: an application-layer protocol for streaming 3D graphics - experimental approach. In *IEEE Int. Conf. on Multimedia and Expo*, volume 1, pages 421–424, 2003.

[ARAR02] Ghassan Al-Regib, Yucel Altunbasak, and Jarek Rossignac. An unequal error protection method for progressively compressed 3D models. In *Int. Conf. On Acoustics, Speech, and Signal Processing*, volume 2, pages 2041–2044, 2002.

[ARAR05] Ghassan Al-Regib, Yucel Altunbasak, and Jarek Rossignac. Error-resilient transmission of 3D models. *ACM Trans. on Graphics*, 2005.

[BCF+05] Stephan Behrendt, Carsten Colditz, Oliver Franzke, Johannes Kopf, and Oliver Deussen. Realistic real-time rendering of landscapes using billboard clouds. *Computer Graphics Forum*, 24(3):507–516, September 2005.

[BCG+96] Gill Barequet, Bernard Chazelle, Leonidas Guibas, Joseph Mitchell, and Ayellet Tal. BOXTREE: A Hierarchical Representation for Surfaces in 3D. *Computer Graphics Forum*, 15(3):387–396, 1996.

[BHGS06] Tamy Boubekeur, Wolfgang Heidrich, Xavier Granier, and Christophe Schlick. Volume-Surface Trees. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2006)*, 25(3):399–406, 2006.

[BHS+08] Jean Botev, Alexander Hohfeld, Hermann Schloss, Ingo Scholtes, Peter Sturm, and Markus Esch. The HyperVerse: concepts for a federated and Torrent-based 3D Web. *Int. J. Adv. Media Commun.*, 2(4), 2008.

[BK02a] Stephan Bischoff and Leif Kobbelt. Streaming 3D Geometry Data Over Lossy Communication Channels. *IEEE International Conference on Multimedia and Expo Proceedings*, 2002.

[BK02b] Stephan Bischoff and Leif Kobbelt. Towards Robust Broadcasting of Geometry Data. *Computers & Graphics*, 26(5):665–675, 2002.

[BKOS97] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational geometry: algorithms and applications*. Springer-Verlag New York, Inc., 1997.

[Blo85] Jules Bloomenthal. Modeling the mighty maple. *ACM Computer Graphics (SIGGRAPH'85)*, 19(3):305–311, July 1985.

[BMG06] Frédéric Boudon, Alexandre Meyer, and Christophe Godin. Survey on Computer Representations of Trees for Realistic and Efficient Rendering. Technical report, LIRIS UMR 5205 CNRS, 2006.

[BNM+08] Antoine Bouthors, Fabrice Neyret, Nelson Max, Eric Bruneton, and Cyril Crassin. Interactive multiple anisotropic scattering in clouds. In *ACM Symposium on Interactive 3D Graphics and Games (I3D)*, 2008.

[Bou04] Frédéric Boudon. *Représentation géométrique multi-échelles de l'architecture des plantes*. PhD thesis, Université de Montpellier II, 2004. In French.

[BPP95] Gavin Bell, Anthony Parisi, and Mark Pesce. The Virtual Reality Modeling Language, Version 1.0 Specification. Web3D Consortium, 1995.

[BSK04] Mario Botsch, Michael Spernat, and Leif Kobbelt. Phong Splatting. *Symposium on Point-Based Graphics*, pages 25–32, 2004.

[BW05] Xavier Baele and Nadine Warzee. Real Time L-System Generated Trees Based on Modern Graphics Hardware. In *SMI'05: Proceedings of the International Conference on Shape Modeling and Applications 2005*, pages 186–195, 2005.

[BWX95] Przemyslaw Bogacki, Stanley E. Weinstein, and Yuesheng Xu. Degree reduction of Bézier curves by uniform approximation with endpoint interpolation. *Computer-Aided Design*, 27(9):651–661, 1995.

[Cam90] Stephen Cameron. Collision Detection by Four–Dimensional Intersectin Testing. *IEEE Transaction on Robotics and Automation*, 6(3):291–302, 1990.

[CBB05] Zhihua Chen, Fritz Barnes, and Bobby Bodenheimer. Hybrid and forward error correction transmission techniques for unreliable transport of 3D geometry. *Multimedia Systems*, 10(3):230–244, 2005.

[CBPEZ04] Liang Cheng, Anusheel Bhushan, Renato Pajarola, and Magda El Zarki. Real-time 3D Graphics Streaming Using MPEG-4. In *BroadWise'04, San Jose, CA, USA*, 2004.

[CBR06] Romain Cavagna, Christian Bouville, and Jerome Royan. P2P Network for very large virtual environment. In *VRST 06: Proceedings of the ACM symposium on Virtual reality software and technology*, 2006.

[CJD+06] Zhi-Quan Cheng, Shi-Yao Jin, Gang Dang, Tao Yang, and Tong Wu. A Service-Oriented Architecture for Progressive Delivery and Adaptive Rendering of 3D Content. In *Proceedings of the 12th International Conference on Virtual Systems and Multimedia (VSMM)*, 2006.

[CN01] Baoquan Chen and Minh Xuan Nguyen. POP: a hybrid point and polygon rendering system for large data. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 45–52. IEEE Computer Society, 2001.

[CO08] Wei Cheng and Wei Tsang Ooi. Receiver-driven View Dependent Streaming of Progressive Mesh. In *Proceedings of the 18th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, 2008.

[COM+07] Wei Cheng, Wei Tsang Ooi, Sebastien Mondet, Romulus Grigoras, and Géraldine Morin. An analytical model for progressive mesh streaming. In *The 15th international conference on Multimedia, ACM, Augsburg, Germany, 24/09/07-29/09/07*, pages 737–746. ACM Press, 2007.

[COM+09] Wei Cheng, Wei Tsang Ooi, Sebastien Mondet, Romulus Grigoras, and Géraldine Morin. Modeling Progressive Mesh Streaming: Does Data Dependency Matter? Submitted to ACM TOMCCAP, 2009.

[COZ98] Daniel Cohen-Or and Eyal Zadicario. Visibility Streaming for Network-based Walkthroughs. In *Graphics Interface*, pages 1–7, June 1998.

[CS06] Michael Casey and Malcolm Slaney. Song Intersection by Approximate Nearest Neighbor Search. In *Proceedings of the 7th International Conference on Music Information Retrieval*, pages 144–149, 2006.

[CSKG03] Evelyne Costes, Hervé Sinoquet, Jean-Jacques Kelner, and Christophe Godin. Exploring within-tree architectural development of two apple treecultivars over 6 years. *Annals of Botany*, 91:91–104, 2003.

[CYB08] Jerry Chen, Ilmi Yoon, and Wes Bethel. Interactive, Internet Delivery of Visualization via Structured Prerendered Multiresolution Imagery. *IEEE Transactions on Visualization and Computer Graphics*, 2008.

[DCSD02] Oliver Deussen, Carsten Colditz, Marc Stamminger, and George Drettakis. Interactive visualization of complex plant ecosystems. In *Proceedings of IEEE Visualization*, October 2002.

[DG00a] Vasilios Darlagiannis and Nicolas Georganas. Virtual Collaboration and Media Sharing using COSMOS. In *Proc. of the 4th World Multiconference on Circuits, Systems, Communications and Computers (CSCC 2000), Athens, Greece*, 2000.

[DG00b] Olivier Devillers and Pierre-Marie Gandoin. Geometric Compression for Interactive Transmission. Technical report, INRIA - Sophia Antipolis, 2000.

[DL05] Oliver Deussen and Bernd Lintermann. *Digital Design of Nature: Computer Generated Plants and Organics*. Springer-Verlag, 2005.

[DN04] Philippe Decaudin and Fabrice Neyret. Rendering Forest Scenes in Real-Time. In *Proceedings of the 15th Eurographics Symposium on Rendering*, pages 93–102, June 2004.

[Eic06] Alexander Eichhorn. Modelling dependency in multimedia streams. In *MULTIMEDIA '06: Proceedings of the 14th annual ACM international conference on Multimedia*, pages 941–950, New York, NY, USA, 2006.

[FACOS03] Shachar Fleishman, Marc Alexa, Daniel Cohen-Or, and Claudio T. Silva. Progressive Point set surfaces. *ACM Transactions on Computer Graphics*, 22(4), 2003.

[Far02] Gerald Farin. *Curves and surfaces for CAGD: a practical guide*. Morgan Kaufmann Publishers Inc., 2002.

[FDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer graphics: principles and practice (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[FGM+99] Roy Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. RFC 2616 - Hypertext Transfer Protocol - HTTP/1.1. Internet Engineering Task Force, Request For Comments, 1999.

[FHL+05] Doug Freimuth, Elbert Hu, Jason LaVoie, Ronald Mraz, Erich Nahum, Prashant Pradhan, and John Tracey. Server network scalability and TCP offload. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 15–15, 2005.

[FJ93] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1(4), 1993.

[FJJ03] Jon Ferraiolo, Fujisawa Jun Jun, and Dean Jackson. Scalable Vector Graphics (SVG) 1.1 Specification. World Wide Web Consortium Recommendation, 2003.

[FK06] Sally Floyd and Eddie Kohler. RFC 4341 - Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 2: TCP-like Congestion Control. Internet Engineering Task Force, Request For Comments, 2006.

[FKMP03] Pavol Federl, Radoslaw Karwowski, Radomir Mech, and Przemyslaw Prusinkiewicz. L-systems and Beyond. Course Notes, SIGGRAPH 03, 2003.

[FKP06] Sally Floyd, Eddie Kohler, and Jitendra Padhye. RFC 4342 - Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC). Internet Engineering Task Force, Request For Comments, 2006.

[Gar00] Jacques Garrigue. Code reuse through polymorphic variants. In *In Workshop on Foundations of Software Engineering*, 2000.

[GBP04] Gael Guennebaud, Loïc Barthe, and Mathias Paulin. Deferred Splatting. *Computer Graphics Forum*, 23(3), 2004.

[GCD02] Romulus Grigoras, Vincent Charvillat, and Matthijs Douze. Optimizing Hypervideo Navigation using a Markov Decision Process Approach. In *MM '02: Proceeding of the 12th ACM international conference on Multimedia*, 2002.

[GDO00] Fabio Ganovelli, John Dingliana, and Carol O'Sullivan. BucketTree: Improving Collision Detection Between Deformable Objects. *Spring Conference in Computer Graphics*, 2000.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[GM03] Laurent Grisoni and Damien Marchal. High Performance generalized cylinders visualization. In *SMI'03: Proceedings of the Shape Modeling International 2003*, page 257, 2003.

[GMW04] Callum Galbraith, Peter MacMurchy, and Brian Wyvill. BlobTree Trees. In *Proceedings of Computer Graphics International*, pages 78–85, June 2004.

[GO05] Yan Gu and Wei Tsang Ooi. Packetization of 3D Progressive Meshes for Streaming over Lossy Networks. In *Proceedings of the 14th International Conference on Computer Communications and Networks (ICCCN)*, 2005.

[Hav04] Herman Johannes Haverkort. *Results on geometric networks and data structures*. PhD thesis, Proefschrift Universiteit Utrecht, 2004.

[HG02] Mojtaba Hosseini and Nicolas Georganas. MPEG-4 BIFS streaming of large virtual environments and their animation on the web. In *Web3D '02: Proceeding of the seventh international conference on 3D Web technology*, pages 19–25. ACM Press, 2002.

[HK02] Albert Harris and Robin Kravets. The design of a transport protocol for on-demand graphical rendering. In *NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 43–49, 2002.

[Hop96] Hugues Hoppe. Progressive meshes. In *SIGGRAPH'96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 99–108, 1996.

[Hop97] Hugues Hoppe. View-dependent refinement of progressive meshes. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 189–198, 1997.

[Hu06] Shun-Yun Hu. A case for 3D streaming on peer-to-peer networks. In *Web3D 06: Proceedings of the eleventh international conference on 3D web technology*, 2006.

[Huf52] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9), 1952.

[Joh67] Stephen C. Johnson. Hierarchical Clustering Schemes. *Psychometrika*, 2:241–254, 1967.

[JSLB00] Linan Jiang, Betty Salzberg, David B. Lomet, and Manuel Barrena. The BT-tree: A Branched and Temporal Access Method. In *The VLDB Journal*, pages 451–460, 2000.

[JW00] Prasad Jogalekar and Murray Woodside. Evaluating the scalability of distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, 11(6), 2000.

[KB04] Leif Kobbelt and Mario Botsch. A Survey of Point-Based Techniques in Computer Graphics. *Computers and Graphics*, 28(6):801–814, 2004.

[KHF06] Eddie Kohler, Mark Handley, and Sally Floyd. RFC 4340 - Datagram Congestion Control Protocol (DCCP). Internet Engineering Task Force, Request For Comments, 2006.

[KHM+98] James T. Klosowski, Martin Held, Joseph Mitchell, Henry Sowizral, and Karel Zikan. Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, 1998.

[Kle08] John Klensin. RFC 5321 - Simple Mail Transfer Protocol. Internet Engineering Task Force, Request For Comments, 2008.

[KLK04] Junho Kim, Seungyong Lee, and Leif Kobbelt. View-Dependent Streaming of Progressive Meshes. *Shape Modeling Applications*, 2004.

[LAM01] Thomas Larsson and Tomas Akenine-Moller. Collision Detection for Continuously Deforming Bodies. In *Eurographics 2001, Short Presentations*, 2001.

[LAM03] Thomas Larsson and Tomas Akenine-Möller. Strategies for Bounding Volume Hierarchy Updates for Ray Tracing of Deformable Models. Technical report, Mälardalen Real-Time Research Centre, February 2003.

[Ler90] Xavier Leroy. The ZINC experiment : an economical implementation of the ML language. Technical Report RT-0117, INRIA, 1990.

[Ler00] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3), 2000.

[LGS+00] Francis C. Li, Anoop Gupta, Elizabeth Sanocki, Li-wei He, and Yong Rui. Browsing digital video. In *CHI'00: Proceedings of the SIGCHI conference on Human factors in computing systems*, 2000.

[LW85] Marc Levoy and Turner Whitted. The Use of Points as a Display Primitive. Technical report, Computer Science Department, University of North Carolina at Chapel Hill, 1985.

[LZS+03] Fabrizio Lamberti, Claudio Zunino, Andrea Sanna, Antonino Fiume, and Marco Maniezzo. An accelerated remote graphics architecture for PDAS. In *Web3D '03: Proceedings of the eighth international conference on 3D Web technology*, 2003.

[MAK96] Farzin Mokhtarian, Sadegh Abbasi, and Josef Kittler. Robust and efficient shape indexing through curvature scale space. In *Proceedings of British Machine Vision Conference*, pages 53–62, 1996.

[Mar00] Ioana M. Martin. Adaptive Rendering of 3D Models over Networks Using Multiple Modalities. Technical report, IBM T. J. Watson Research Center, 2000.

[MBGB00] Michael Marcellin, Ali Bilgin, Michael Gormish, and Martin Boliek. An Overview of JPEG-2000. *Data Compression Conference*, 0, 2000.

[MBM+07] Sebastien Mondet, Frédéric Boudon, Géraldine Morin, Romulus Grigoras, and Mathias Paulin. Compression progressive de modèles de plantes à base de cylindres généralisés. In *Association Française d'Informatique Graphique (AFIG), Marne-la-vallée, France, 26/11/07-28/11/07*, 2007. in French.

[MCM+08] Sebastien Mondet, Wei Cheng, Géraldine Morin, Romulus Grigoras, Frédéric Boudon, and Wei Tsang Ooi. Streaming of Plants in Distributed Virtual Environments. In *MM '08: Proceeding of the 16th ACM international conference on Multimedia*, pages 1–10. ACM, 2008. Best Paper Award.

[MCM+09] Sebastien Mondet, Wei Cheng, Géraldine Morin, Romulus Grigoras, Frédéric Boudon, and Wei Tsang Ooi. Compact and Progressive Plant Models for Streaming in Networked Virtual Environments. To appear in ACM TOMCCAP, 2009.

[MdR99] Todd C. Miller and Theo de Raadt. strlcpy and strlcat: consistent, safe, string copy and concatenation. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, 1999.

[MMG05] Sebastien Mondet, Géraldine Morin, and Romulus Grigoras. Mise en ligne de modèles 3D echelonables basés points. In *Association Française d'Informatique Graphique (AFIG), Strasbourg, France, 28/11/05-30/11/05*. AFIG, 2005. in French.

[MMG07] Sebastien Mondet, Géraldine Morin, and Romulus Grigoras. Optimized Box-Trees For Server-side Viewpoint Culling On Large 3D Scenes. In *Groupe de Travail en Modélisation Géométrique, Valenciennes, France, 21/03/07-22/03/07*, 2007.

[MNP01] Alexandre Meyer, Fabrice Neyret, and Pierre Poulin. Interactive Rendering of Trees with Shading and Shadows. In *Proceedings of the Eurographics Workshop on Rendering Techniques*, pages 183–196, June 2001.

[MSL99] Rakesh Mohan, John R. Smith, and Chung-Sheng Li. Adapting multimedia Internet content for universal access. *Multimedia, IEEE Transactions on*, 1(1), 1999.

[MW08] Yaron Minksy and Stephen Weeks. Caml trading - experiences with functional programming on Wall Street. *Journal of Functional Programming*, 18(04), 2008.

[MZ03] Fang Meng and Hongbin Zha. Streaming Transmission of Point-Sampled Geometry Based on View-Dependent Level-of-Detail. *International Conference on 3D Digital Imaging and Modeling*, 2003.

[NCO03] Yuval Noimark and Daniel Cohen-Or. Streaming Scenes to MPEG-4 Video-Enabled Devices. *IEEE Computer Graphics and Applications*, 23(1):58–64, 2003.

[NFD07] Boris Neubert, Thomas Franken, and Oliver Deussen. Approximate image-based tree-modeling using particle flows. *ACM Transactions on Graphics*, 26(3):88, July 2007.

[OP98] Stephan Olbrich and Helmut Pralle. High-Performance Online Presentation of Complex 3D Scenes. In *HPN 98: Proceedings of the IFIP TC-6 Eigth International Conference on High Performance Networking*, pages 471–484, 1998.

[OP99] Stephan Olbrich and Helmut Pralle. Virtual reality movies-real-time streaming of 3D objects. *Comput. Networks*, 31(21), 1999.

[Oss86] Robert Osserman. *A Survey of Minimal Surfaces*. Dover Publications, New York, 1986.

[PA00] Vern Paxson and Marc Allman. RFC 2988 - Computing TCP's Retransmission Timer. Internet Engineering Task Force, Request For Comments, 2000.

[Pau03] Mark Pauly. *Point Primitives for Interactive Modeling and Processing of 3D Geometry*. PhD thesis, Federal Institute of Technology (ETH) of Zurich, 2003.

[PCG07] Cezar Plesca, Vincent Charvillat, and Romulus Grigoras. Adapting Content Delivery to Observable Resources and Semi-Observable User Interest. Technical Report IRIT/RR–2007-3–FR, IRIT, 2007.

[PCG08] Cezar Plesca, Vincent Charvillat, and Romulus Grigoras. User-aware adaptation by subjective metadata and inferred implicit descriptors. In Marc Spaniol, editor, *Multimedia Semantics-The Role of Metadata*, volume 101-2008. Springer, 2008.

[PDKB+08] Christophe Pradal, Samuel Dufour-Kowalski, Frédéric Boudon, Christian Fournier, and Christophe Godin. OpenAlea: A visual programming and component-based software platform for plant modeling. *Functional Plant Biology*, 35, 2008.

[PKL06] Sung-Bum Park, Chang-Su Kim, and Sang Uk Lee. Error Resilient 3-D Mesh Compression. *IEEE Transactions on Multimedia*, 8:885–895, 2006.

[PL90] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer Verlag, 1990.

[PMKL01] Przemyslaw Prusinkiewicz, Lars Mündermann, Radoslaw Karwowski, and Brendan Lane. The use of positional information in the modeling of plants. *ACM Computer Graphics (SIGGRAPH'01)*, 22(4):289–300, 2001.

[Pos80] Jon Postel. RFC 768 - User Datagram Protocol. Internet Engineering Task Force, Request For Comments, 1980.

[Pos81] Jon Postel. RFC 793 - Transmission Control Protocol Specification. Internet Engineering Task Force, Request For Comments, 1981.

[Pru86] Przemyslaw Prusinkiewicz. Graphical applications of L-systems. In *Vision Interface*, pages 247–253, May 1986.

[RCB+02] Inmaculada Remolar, Miguel Chover, Oscar Belmonte, José Ribelles, and Cristina Rebollo. Geometric Simplification of Foliage. In *Eurographics'02 Short Presentations*, pages 397–404, 2002.

[RGCB07] Jerome Royan, Patrick Gioia, Romain Cavagna, and Christian Bouville. Network-Based Visualization of 3D Landscapes and City Models. *IEEE Comput. Graph. Appl.*, 27(6), 2007.

[Ric03] Iain E. Richardson. *H.264 and MPEG-4 Video Compression: Video Coding for Next Generation Multimedia*. Wiley, 2003.

[RL00] Szymon Rusinkiewicz and Marc Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. In Kurt Akeley, editor,

Siggraph 2000, Computer Graphics Proceedings, pages 343–352. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[RL01] Szymon Rusinkiewicz and Marc Levoy. Streaming QSplat: a viewer for networked visualization of large, dense models. In *Symposium on Interactive 3D Graphics*, pages 63–68, 2001.

[Scu01] Scut. Exploiting format string vulnerabilities. TESO Security Group, 2001.

[Sen02] Daniel Senie. Network Address Translator (NAT)-Friendly Application Design Guidelines. Internet Engineering Task Force, Request For Comments, 2002.

[SFK08] Pyda Srisuresh, Bryan Ford, and Dan Kegel. State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs). Internet Engineering Task Force, Request For Comments, 2008.

[SHJ08] Wei-Lun Sung, Shun-Yun Hu, and Jehn-Ruey Jiang. Selection Strategies for Peer-to-Peer 3D Streaming. In *Proc. 18th International workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 2008.

[SRG97] Hervé Sinoquet, Pierre Rivet, and Christophe Godin. Assessment of the three-dimensional architecture of walnut trees using digitising. *Silva Fennica*, 31(3):265–273, 1997.

[SSB04] Jorg Sahm, Ingo Soetebier, and Horst Birthelmer. Efficient representation and streaming of 3D scenes. *Computers and Graphics*, 28(1):15–24, 2004.

[TA04] Dihong Tian and Ghassan AlRegib. FQM: a fast quality measure for efficient transmission of textured 3D models. In *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*, 2004.

[Tan01] Andrew S. Tanenbaum. *Modern operating systems*. Prentice Hall, 2001.

[Tan02] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 2002.

[Tau99] Gabriel Taubin. 3D Geometry Compression and Progressive Transmission. Eurographics State of the Art Report, 1999.

[TL01] Eyal Teler and Dani Lischinski. Streaming of Complex 3D Scenes for Remote Walkthroughs. *Computer Graphics Forum*, 20(3), 2001.

[TYOC05] Bugra Tari, Yucel Yemez, Oznur Ozkasap, and Reha Civanlar. Progressive View-Dependent Transmission of 3D Models over Lossy Netowrk. In *Proceedings of the 13th European Signal Proceesing Conference*, 2005.

[vdB97] Gino van den Bergen. Efficient Collision Detection of Complex Deformable Models using AABB Trees. *Journal of Graphics Tools: JGT*, 2(4):1–14, 1997.

[WMS06] Sven Woop, Gerd Marmitt, and Philipp Slusallek. B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware*, 2006.

[WP95] Jason Weber and Joseph Penn. Creation and Rendering of Realistic Trees. *ACM Computer Graphics (SIGGRAPH'95)*, 29(3):119–128, August 1995.

[YKK01] Zhidong Yan, Sunil Kumar, and Jay Kuo. Error resilient coding of 3D graphic models via adaptive mesh segmentation. *IEEE Trans. Circuits Syst. Video Technol*, 11:860–873, 2001.

[YLK04] Sheng Yang, Chao-Hua Lee, and C.-C. Jay Kuo. Optimized mesh and texture multiplexing for progressive textured model transmission. In *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*, 2004.

[YM06] Sung-Eui Yoon and Dinesh Manocha. Cache-Efficient Layouts of Bounding Volume Hierarchies. In *Eurographics*, 2006.

[YNBH09] Qizhi Yu, Fabrice Neyret, Eric Bruneton, and Nicolas Holzschuch. Scalable Real-Time Animation of Rivers. *Comput. Graph. Forum*, 28(2), 2009.

[ZBJ06] Xiaopeng Zhang, Frédéric Blaise, and Marc Jaeger. Multiresolution plant models with complex organs. In *VRCIA'06: Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, pages 331–334, 2006.

[ZPKG02] Matthias Zwicker, Mark Pauly, Oliver Knoll, and Markus Gross. Pointshop 3D: An Interactive System for Point-Based Surface Editing. In *SIGGRAPH 2002*, pages 322–329, 2002.