Dynamic Adaptive 3D Streaming over HTTP

For the University of Toulouse PhD granted by the INP Toulouse
Presented and defended on Friday 29th November, 2019 by Thomas Forgione

Gilles GESQUIÈRE, president
Sidonie CHRISTOPHE, reviewer
Gwendal SIMON, reviewer
Maarten WIJNANTS, examiner
Wei Tsang OOI, examiner
Vincent CHARVILLAT, thesis supervisor
Axel CARLIER, thesis co-supervisor
Géraldine MORIN, thesis co-supervisor

Doctoral school and field: EDMITT: École Doctorale de Mathématiques, Informatiques et Télécommunications de Toulouse
Field: Computer science and telecommunication
Research unit: IRIT (5505)
Thesis supervisors: Vincent CHARVILLAT, Axel CARLIER and Géraldine MORIN
Reviewers: Sidonie CHRISTOPHE and Gwendal SIMON

Titre : Transmission Adaptative de Modèles 3D Massifs

Résumé :
Avec les progrès de l'édition de modèles 3D et des techniques de reconstruction 3D, de plus en plus de modèles 3D sont disponibles et leur qualité augmente. De plus, le support de la visualisation 3D sur le web s'est standardisé ces dernières années. Un défi majeur est donc de transmettre des modèles massifs à distance et de permettre aux utilisateurs de visualiser et de naviguer dans ces environnements virtuels. Cette thèse porte sur la transmission et l'interaction de contenus 3D et propose trois contributions majeures.

Tout d'abord, **nous développons une interface de navigation dans une scène 3D avec des signets** – de petits objets virtuels ajoutés à la scène sur lesquels l'utilisateur peut cliquer pour atteindre facilement un emplacement recommandé. Nous décrivons une étude d'utilisateurs où les participants naviguent dans des scènes 3D avec ou sans signets. Nous montrons que les utilisateurs naviguent (et accomplissent une tâche donnée) plus rapidement en utilisant des signets. Cependant, cette navigation plus rapide a un inconvénient sur les performances de la transmission : un utilisateur qui se déplace plus rapidement dans une scène a besoin de capacités de transmission plus élevées afin de bénéficier de la même qualité de service. Cet inconvénient peut être atténué par le fait que les positions des signets sont connues à l'avance : en ordonnant les faces du modèle 3D en fonction de leur visibilité depuis un signet, on optimise la transmission et donc, on diminue la latence lorsque les utilisateurs cliquent sur les signets.

Deuxièmement, **nous proposons une adaptation du standard de transmission DASH (Dynamic Adaptive Streaming over HTTP), très utilisé en vidéo, à la transmission de maillages texturés 3D**. Pour ce faire, nous divisons la scène en un arbre k-d où chaque cellule correspond à un adaptation set DASH. Chaque cellule est en outre divisée en segments DASH d'un nombre fixe de faces, regroupant des faces de surfaces comparables. Chaque texture est indexée dans son propre adaptation set à différentes résolutions. Toutes les métadonnées (les cellules de l'arbre k-d, les résolutions des textures, etc.) sont référencées dans un fichier XML utilisé par DASH pour indexer le contenu: le MPD (Media Presentation Description). Ainsi, notre framework hérite de la scalabilité offerte par DASH. Nous proposons ensuite des algorithmes capables d'évaluer l'utilité de chaque segment de données en fonction du point de vue du client, et des politiques de transmission qui décident des segments à télécharger.

Enfin, **nous étudions la mise en place de la transmission et de la navigation 3D sur les appareils mobiles**. Nous intégrons des signets dans notre version 3D de DASH et proposons une version améliorée de notre client DASH qui bénéficie des signets. Une étude sur les utilisateurs montre qu'avec notre politique de chargement adaptée aux signets, les signets sont plus susceptibles d'être cliqués, ce qui améliore à la fois la qualité de service et la qualité d'expérience des utilisateurs.

Title: Dynamic Adaptive 3D Streaming over HTTP

Abstract:
With the advances in 3D models editing and 3D reconstruction techniques, more and more 3D models are available and their quality is increasing. Furthermore, the support of 3D visualization on the web has become standard during the last years. A major challenge is thus to deliver these remote heavy models and to allow users to visualise and navigate in these virtual environments. This thesis focuses on 3D content streaming and interaction, and proposes three major contributions.

First, **we develop a 3D scene navigation interface with bookmarks** – small virtual objects added to the scene that the user can click on to ease reaching a recommended location. We describe a user study where participants navigate in 3D scenes with and without bookmarks. We show that users navigate (and accomplish a given task) faster when using bookmarks. However, this faster navigation has a drawback on the streaming performance: a user who moves faster in a scene requires higher streaming capabilities in order to enjoy the same quality of service. This drawback can be mitigated using the fact that bookmarks positions are known in advance: by ordering the faces of the 3D model according to their visibility at a bookmark, we optimize the streaming and thus, decrease the latency when users click on bookmarks.

Secondly, **we propose an adaptation of Dynamic Adaptive Streaming over HTTP (DASH), the video streaming standard, to 3D textured meshes streaming**. To do so, we partition the scene into a k-d tree where each cell corresponds to a DASH adaptation set. Each cell is further divided into DASH segments of a fixed number of faces, grouping together faces of similar areas. Each texture is indexed in its own adaptation set, and multiple DASH representations are available for different resolutions of the textures. All the metadata (the cells of the k-d tree, the resolutions of the textures, etc.) is encoded in the Media Presentation Description (MPD): an XML file that DASH uses to index content. Thus, our framework inherits DASH scalability. We then propose clients capable of evaluating the usefulness of each chunk of data depending on their viewpoint, and streaming policies that decide which chunks to download.

Finally, **we investigate the setting of 3D streaming and navigation on mobile devices**. We integrate bookmarks in our 3D version of DASH and propose an improved version of our DASH client that benefits from bookmarks. A user study shows that with our dedicated bookmark streaming policy, bookmarks are more likely to be clicked on, enhancing both users quality of service and quality of experience.

Acknowledgments

First of all, I would like to thank my advisors, Vincent Charvillat, Axel Carlier, and Géraldine Morin for luring me into doing a PhD (which was a lot of work), for the support, and for the fun (and beers[1]). I took *a little time* to take the decision of starting a PhD, so I also want to thank them for their patience. I also want to thank Wei Tsang Ooi, for the ideas, the discussions and for the polish during the deadlines.

Then, I want to thank Sidonie Christophe and Gwendal Simon for reviewing this manuscript: I appreciated the feedback and constructive comments. I also want to thank all the members of the jury, for their attention and the interesting discussions during the defense.

I want to thank all the kids of the lab and elsewhere that contributed to the mood (and beers[2]): Bastien, Vincent, Julien, Sonia, Matthieu, Jean, Damien, Richard, Thibault, Clément, Arthur, Thierry, the other Matthieu, the other Julien, Paul, Maxime, Quentin, Adrian, Salomé. I also want to thank Praveen: working with him was a pleasure.

I would also like to thank the big ones (whom I forgot to thank during the defense, *oopsies*), Sylvie, Jean-Denis, Simone, Yvain, Pierre. They, as well as my advisors, not only helped during my PhD, but they also were my teachers back in engineering school and are a great part of the reason why I enjoyed being in school and being a PhD student.

Then, I also want to thank Sylvie and Muriel, for the administrative parts of the PhD, which can often be painful.

I would also like to thank the colleagues from when I was in engineering school, since they contributed to the skills that I used during this PhD: Alexandre, Killian, David, Maxence, Martin, Maxime, Korantin, Marion, Amandine, Émilie.

Finally, I want to thank my brother, my sister and my parents, for the support and guidance. They have been decisive to my education and I would not be writing this today if it was not for them.

[1] drink responsibly

[2] I mean, seriously, drink responsibly

Contents

Introduction

During the last years, 3D acquisition and modeling techniques have made tremendous progress. Recent software uses 2D images from cameras to reconstruct 3D data, e.g. Meshroom[1] is a free and open source software which got almost 200 000 downloads on fosshub[2], which use *structure-from-motion* and *multi-view-stereo* to infer a 3D model. More and more devices are specifically built to harvest 3D data: for example, LIDAR (Light Detection And Ranging) can compute 3D distances by measuring time of flight of light. The recent research interest for autonomous vehicles allowed more companies to develop cheaper LIDARs, which increase the potential for new 3D content creation. Thanks to these techniques, more and more 3D data become available. These models have potential for multiple purposes, for example, they can be printed, which can reduce the production cost of some pieces of hardware or enable the creation of new objects, but most uses are based on visualization. For example, they can be used for augmented reality, to provide user with feedback that can be useful to help worker with complex tasks, but also for fashion (for example, Fittingbox[3] is a company that develops software to virtually try glasses, as in Figure 1).

Figure 1: My face with augmented glasses

[1] https://alicevision.org/#meshroom
[2] https://www.fosshub.com/Meshroom.html
[3] https://www.fittingbox.com

3D acquisition and visualization is also useful to preserve cultural heritage, and software such as Google Heritage or 3DHop are such examples, or to allow users navigating in a city (as in Google Earth or Google Maps in 3D). Sketchfab[4] (see Figure 2) is an example of a website allowing users to share their 3D models and visualize models from other users.

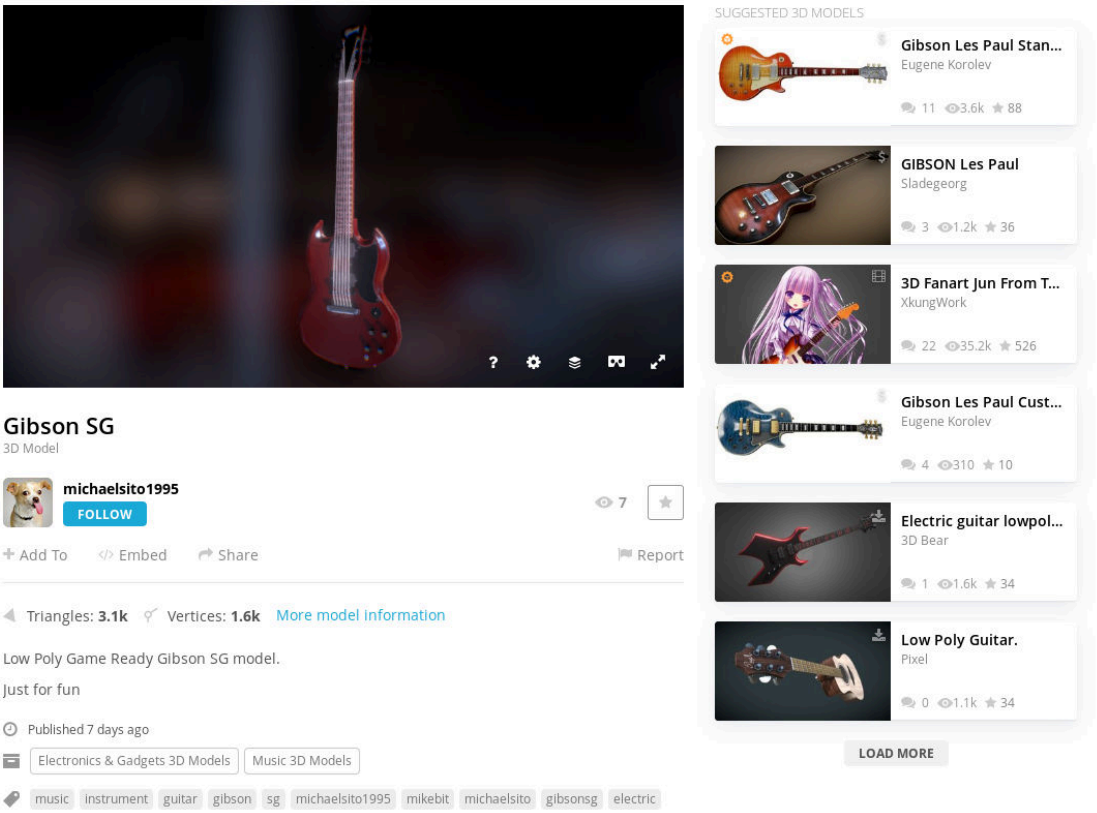


Figure 2: Sketchfab interface

In most 3D visualization systems, the 3D data are stored on a server and need to be transmitted to a terminal before the user can visualize them. The improvements in the acquisition setups we described lead to an increasing quality of the 3D models, thus an increasing size in bytes as well. Simply downloading 3D content and waiting until it is fully downloaded to let the user visualize it is no longer a satisfactory solution, so adaptive streaming is needed. In this thesis, we propose a full framework for navigation and streaming of large 3D scenes, such as districts or whole cities.

[4] https://sketchfab.com

1 Open problems

The objective of our work is to design a system which allows a user to access remote 3D content. A 3D streaming client has lots of tasks to accomplish:

- Decide what part of the content to download next,
- Download the next part,
- Parse the downloaded content,
- Add the parsed result to the scene,
- Render the scene,
- Manage the interaction with the user.

This opens multiple problems which need to be considered and will be studied in this thesis.

Content preparation. Before streaming content, it needs to be prepared. The segmentation of the content into chunks is particularly important for streaming since it allows transmitting only a portion of the data to the client. The downloaded chunks can be rendered while more chunks are being downloaded. Content preparation also includes compression. One of the questions this thesis has to answer is: *what is the best way to prepare 3D content so that a streaming client can progressively download and render the 3D model?*

Streaming policies. Once our content is prepared and split in chunks, a client needs to determine which chunks should be downloaded first. A chunk that contains data in the field of view of the user is more relevant than a chunk that is not inside; a chunk that is close to the camera is more relevant than a chunk far away from the camera. This should also include other contextual parameters, such as the size of a chunk, the bandwidth and the user's behaviour. In order to propose efficient streaming policies, we need to know *how to estimate a chunk utility, and how to determine which chunks need to be downloaded depending the user's interactions?*

Evaluation. In such systems, two commonly used criteria for evaluation are quality of service, and quality of experience. The quality of service is a network-centric metric, which considers values such as throughput and measures how well the content is served to the client. The quality of experience is a user-centric metric: it relies on user perception and can only be measured by asking how users feel about a system. To be able to know which streaming policies are best, one needs to know *how to compare streaming policies and evaluate the impact of their parameters on the quality of service of the streaming system and on the quality of experience of the final user?*

Implementation. The objective of our work is to setup a client-server architecture that answers the above problems: content preparation, chunk utility, streaming policies. In this regard, we have to find out *how do we build this architecture that keeps a low computational load on the server so it scales up and on the client so that it has enough resources to perform the tasks described above?*

2 Thesis outline

First, in Chapter 1, we give some preliminary information required to understand the types of objects we are manipulating in this thesis. We then proceed to compare 3D and video content: video and 3D share many features, and analyzing video setting gives inspiration for building a 3D streaming system.

In Chapter 2, we present a review of the state of the art in multimedia interaction and streaming. This chapter starts with an analysis of the video streaming standards. Then it reviews the different 3D streaming approaches. The last section of this chapter focuses on 3D interaction.

Then, in Chapter 3, we present our first contribution: an in-depth analysis of the impact of the UI on navigation and streaming in a 3D scene. We first develop a basic interface for navigating in 3D and then, we introduce 3D objects called *bookmarks* that help users navigating in the scene. We then present a user study that we conducted on 51 people which shows that bookmarks ease user navigation: they improve performance at tasks such as finding objects. We analyze how the presence of bookmarks impacts the streaming: we propose and evaluate streaming policies based on precomputations relying on bookmarks and that measurably increase the quality of experience.

In Chapter 4, we present the most important contribution of this thesis: DASH-3D. DASH-3D is an adaptation of DASH (Dynamic Adaptive Streaming over HTTP): the video streaming standard, to 3D streaming. We first describe how we adapt the concepts of DASH to 3D content, including the segmentation of content. We then define utility metrics that rate each chunk depending on the user's position. Then, we present a client and various streaming policies based on our utilities which can benefit from DASH format. We finally evaluate the different parameters of our client.

In Chapter 5, we present our last contribution: the integration of the interaction ideas that we developed in Chapter 3 into DASH-3D. We first develop an interface that allows desktop as well as mobile devices to navigate in streamed 3D scenes, and that introduces a new style of bookmarks. We then explain why simply applying the ideas developed in Chapter 3 is not sufficient and we propose more efficient precomputations that enhance the streaming. Finally, we present a user study that provides us with traces on which we evaluate the impact of our extension of DASH-3D on the quality of service and on the quality of experience.

Chapter 1

Foreword

Contents

A 3D streaming system is a system that progressively collects 3D data. The previous chapter voluntarily remained vague about what *3D data* actually are. This chapter presents in detail the 3D data we consider and how they are rendered. We also give insights about interaction and streaming by comparing the 3D setting to the video one.

1.1 What is a 3D model?

1.1.1 3D data

The 3D models we are interested in are sets of textured meshes, which can potentially be arranged in a scene graph. Such models can typically contain the following:

- **Vertices**, which are 3D points,
- **Faces**, which are polygons defined from vertices (most of the time, they are triangles),
- **Textures**, which are images that can be used to paint faces in order to add visual richness,
- **Texture coordinates**, which are information added to a face, describing how the texture should be painted over it,
- **Normals**, which are 3D vectors that can give information about light behaviour on a face.

The Wavefront OBJ is a format that describes all these elements in text format. A 3D model encoded in the OBJ format typically consists in two files: the material file (`.mtl`) and the object file (`.obj`).

The material file declares all the materials that the object file will reference. A material consists in name, and other photometric properties such as ambient, diffuse and specular colors, as well as texture maps, which are images that are painted on faces. Each face corresponds to a material. A simple material file is visible on Snippet 1.2.

The object file declares the 3D content of the objects. It declares vertices, texture coordinates and normals from coordinates (e.g. `v 1.0 2.0 3.0` for a vertex, `vt 1.0 2.0` for a texture coordinate, `vn 1.0 2.0 3.0` for a normal). These elements are numbered starting from 1. Faces are declared by using the indices of these elements. A face is a polygon with an arbitrary number of vertices and can be declared in multiple manners:

- `f 1 2 3` defines a triangle face that joins the first, the second and the third declared vertices;

- `f 1/1 2/3 3/4` defines a similar triangle but with texture coordinates, the first texture coordinate is associated to the first vertex, the third texture coordinate is associated to the second vertex, and the fourth texture coordinate is associated with the third vertex;
- `f 1//1 2//3 3//4` defines a similar triangle but referencing normals instead of texture coordinates;
- `f 1/1/1 2/3/3 3/4/4` defines a triangle with both texture coordinates and normals.

An object file can include materials from a material file (`mtllib path.mtl`) and apply the materials that it declares to faces. A material is applied by using the `usemtl` keyword, followed by the name of the material to use. The faces declared after a `usemtl` are painted using the material in question. An example of object file is visible on Snippet 1.1.

```
1  mtllib cube.mtl
2
3  usemtl cubemtl
4
5  v -0.5 -0.5 -0.5
6  v -0.5 -0.5 0.5
7  v -0.5 0.5 -0.5
8  v -0.5 0.5 0.5
9  v  0.5 -0.5 -0.5
10 v  0.5 -0.5 0.5
11 v  0.5 0.5 -0.5
12 v  0.5 0.5 0.5
13
14 vt 0.0 0.0
15 vt 0.0 1.0
16 vt 1.0 0.0
17 vt 1.0 1.0
18
19 f 1/1 2/3 4/4 3/2
20 f 2/1 6/3 8/4 4/2
21 f 6/1 5/3 7/4 8/2
22 f 5/1 1/3 3/4 7/2
23 f 4/1 8/3 7/4 3/2
24 f 2/1 1/3 5/4 6/2
```

Snippet (1.1) An object file describing a cube

```
1 newmtl cubemtl
2 Ka 1.0 1.0 1.0
3 Kd 1.0 1.0 1.0
4 Ks 1.0 1.0 1.0
5 map_Kd cube.png
```

Snippet (1.2) A material file describing a material

(a) A rendering of the cube

Figure 1.1: The OBJ representation of a cube and its render

1.1.2 Rendering a 3D model

A typical 3D renderer follows Algorithm 1. The first task the renderer needs to perform is sending the data to the GPU: this is done in the loading loop during an initialization

Algorithm 1: A rendering algorithm

```
/* Initialization */
for object ∈ scene do
    load_geometry(object.geometry);
    load_material(object.material);
end

/* Render loop */
while true do
    for object ∈ scene do
        bind_material(object.material);
        draw(object.geometry);
    end
end
```

step. This step can be slow, but it is generally acceptable since it only occurs once at the beginning of the program. Then, the renderer starts the rendering loop: at each frame, it renders the whole scene: for each object, it binds the corresponding material to the GPU and then renders the object. During the rendering loop, there are two things to consider regarding performances:

- the more faces in a geometry, the slower the `draw` call;
- the more objects in the scene, the more overhead caused by the CPU/GPU communication at each step of the loop.

The way the loop works forces objects with different materials to be rendered separately. An efficient renderer keeps the number of objects in a scene low to avoid introducing overhead. However, an important feature of 3D engines regarding performance is frustum culling. The frustum is the viewing volume of the camera. Frustum culling consists in skipping the objects that are outside the viewing volume of the camera in the rendering loop. Algorithm 2 is a variation of Algorithm 1 with frustum culling.

A renderer that uses a single object avoids the overhead, but fails to benefit from frustum culling. An optimized renderer needs to find a compromise between a too fine partition of the scene, which introduces overhead, and a too coarse partition, which introduces useless rendering.

1.2 Similarities and differences between video and 3D

The video streaming setting and the 3D streaming setting share many similarities: at a higher level of abstraction, both systems allow a user to access remote content without

Algorithm 2: A rendering algorithm with frustum culling

```
/* Initialization */
for object ∈ scene do
    load_geometry(object.geometry);
    load_texture(object.texture);
end

/* Render loop */
while true do
    for object ∈ scene do
        if object ∩ camera_frustum ≠ ∅ then
            bind_texture(object.texture);
            draw(object.geometry);
        end
    end
end
```

having to wait until everything is loaded. Analyzing similarities and differences between the video and the 3D scenarios as well as having knowledge about video streaming literature are the key to developing an efficient 3D streaming system.

1.2.1 Chunks of data

In order to be able to perform streaming, data need to be segmented so that a client can request chunks of data and display it to the user while requesting another chunk. In video streaming, data chunks typically consist in a few seconds of video. In mesh streaming, some progressive mesh approaches encode a base mesh that contains low resolution geometry and textures and different chunks that increase the resolution of the base mesh. Otherwise, a mesh can also be segmented by separating geometry and textures, creating chunks that contain some faces of the model, or some other chunks containing textures.

1.2.2 Data persistence

One of the main differences between video and 3D streaming is data persistence. In video streaming, only one chunk of video is required at a time. Of course, most video streaming services prefetch some future chunks, and keep in cache some previous ones, but a minimal system could work without latency and keep in memory only two chunks: the current one and the next one.

Already a few problems appear here regarding 3D streaming:

- depending on the user's field of view, many chunks may be required to perform a single rendering;

- chunks do not become obsolete the way they do in video, a user navigating in a 3D scene may come back to a same spot after some time, or see the same objects but from elsewhere in the scene.

1.2.3 Multiple representations

All major video streaming platforms support multi-resolution streaming. This means that a client can choose the quality at which it requests the content. It can be chosen directly by the user or automatically determined by analyzing the available resources (size of the screen, downloading bandwidth, device performances)

Figure 1.2: The different qualities available for a Youtube video

Similarly, recent work in 3D streaming have proposed different ways to progressively stream 3D models, displaying a low quality version of the model to the user without latency, and supporting interaction with the model while details are being downloaded. Such strategies are reviewed in Section 2.2.

1.2.4 Media types

Just like a video, a 3D scene is composed of different media types. In video, those media are mostly images, sounds, and subtitles, whereas in 3D, those media are geometry or textures. In both cases, an algorithm for content streaming has to acknowledge those different media types and manage them correctly.

In video streaming, most of the data (in terms of bytes) are used for images. Thus, the most important thing a video streaming system should do is to optimize images streaming.

That is why, on a video on Youtube for example, there may be 6 available qualities for images (144p, 240p, 320p, 480p, 720p and 1080p) but only 2 qualities for sound. This is one of the main differences between video and 3D streaming: in a 3D setting, the ratio between geometry and texture varies from one scene to another, and leveraging between those two types of content is a key problem.

1.2.5 Interaction

The ways of interacting with content is another important difference between video and 3D. In a video interface, there is only one degree of freedom: time. The only things a user can do is letting the video play, pausing, resuming, or jumping to another time in the video. There are also controls for other options that are described on this help page[1]. All the keyboard shortcuts are summed up in Figure 1.3. Those interactions are different if the user is using a mobile device.

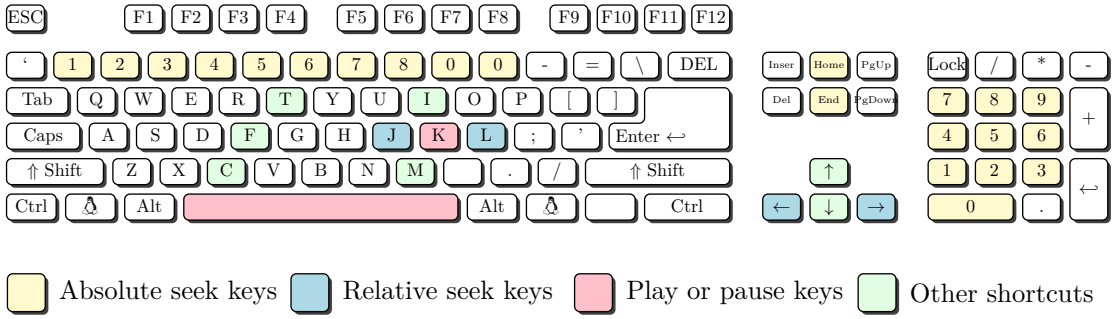


Figure 1.3: Youtube shortcuts (white keys are unused)

When it comes to 3D, there are many approaches to manage user interaction. Some interfaces mimic the video scenario, where the only variable is the time and the camera follows a predetermined path on which the user has no control. These interfaces are not interactive, and can be frustrating to the user who might feel constrained.

Some other interfaces add 2 degrees of freedom to the timeline: the user does not control the camera's position but can control the angle. This mimics the 360 video scenario. This is typically the case of the video game *nolimits 2: roller coaster simulator*[2] which works with VR devices (oculus rift, HTC vive, etc.) where the only interaction available to the user is turning the head.

Finally, most of the other interfaces give at least 5 degrees of freedom to the user: 3 being the coordinates of the camera's position, and 2 being the angles (assuming the up vector is unchangeable, some interfaces might allow that, giving a sixth degree of freedom). The most common controls are the trackball controls where the user rotate the object like

[1] https://web.archive.org/web/20191014131350/https://support.google.com/youtube/answer/7631406?hl=en

[2] http://nolimitscoaster.com/

a ball (live example here)[3] and the orbit controls, which behave like the trackball controls but preserving the up vector (live example here)[4]. These types of controls are notably used on the popular mesh editor MeshLab[5] and SketchFab[6], the YouTube for 3D models.

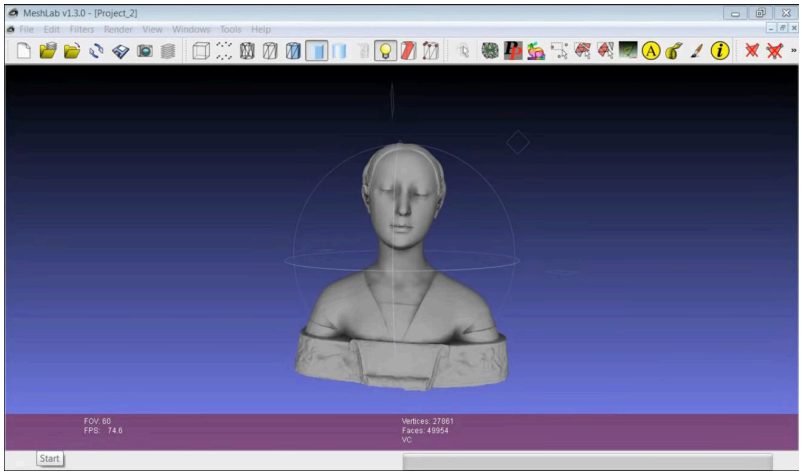


Figure 1.4: Screenshot of MeshLab

Another popular way of controlling a free camera in a virtual environment is the first person controls (live example here)[7]. These controls are typically used in shooting video games, the mouse rotates the camera and the keyboard translates it.

1.2.6 Relationship between interface, interaction and streaming

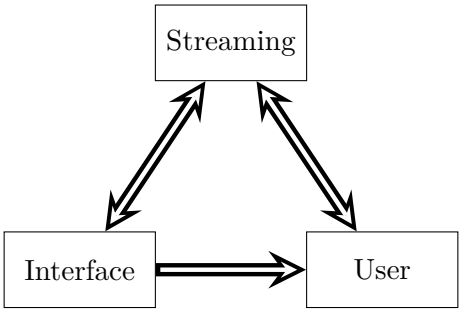
In both video and 3D systems, streaming affects interaction. For example, in a video streaming scenario, if a user sees that the video is fully loaded, they might start moving around on the timeline, but if they see that the streaming is just enough to not stall, they might prefer not interacting and just watch the video. If the streaming stalls for too long, the user might seek somewhere else hoping for the video to resume, or get frustrated and leave the video. The same types of behaviour occur in 3D streaming: if a user is somewhere in a scene, and sees more data appearing, they might wait until enough data have arrived, but if they see nothing happens, they might leave to look for data somewhere else.

Those examples show how streaming can affect interaction, but interaction also affects streaming. In a video streaming scenario, if a user is watching peacefully without interacting, the system just has to request the next chunks of video and display them. However, if a user starts seeking at a different time of the streaming, the streaming would most likely stall until the system is able to gather the data it needs to resume the video. Just

[3] https://threejs.org/examples/?q=controls#misc_controls_trackball
[4] https://threejs.org/examples/?q=controls#misc_controls_orbit
[5] http://www.meshlab.net/
[6] https://sketchfab.com/
[7] https://threejs.org/examples/?q=controls#misc_controls_pointerlock

like in the video setup, the way a user navigates in a networked virtual environment affects the streaming. Moving slowly allows the system to collect and display data to the user, whereas moving frenetically puts more pressure on the streaming: the data that the system requested may be obsolete when the response arrives.

Moreover, the interface and the way elements are displayed to the user also impacts his behaviour. A streaming system can use this effect to enhancing the quality of experience by providing feedback on the streaming to the user via the interface. For example, on Youtube, the buffered portion of the video is displayed in light grey on the timeline, whereas the portion that remains to be downloaded is displayed in dark grey. A user is more likely to click on the light grey part of the timeline than on the dark grey part, preventing the streaming from stalling.



1.3 Implementation details

During this thesis, a lot of software has been developed, and for this software to be successful and efficient, we chose appropriate languages. When it comes to 3D streaming systems, we need two kind of software.

- **Interactive applications** which can run on as many devices as possible so we can easily conduct user studies. For this context, we chose the **JavaScript** language.
- **Native applications** which can run fast on desktop devices, in order to prepare data, run simulations and evaluate our ideas. For this context, we chose the **Rust** language.

1.3.1 JavaScript

THREE.js. On the web browser, it is now possible to perform 3D rendering by using WebGL. However, WebGL is very low level and it can be painful to write code, even to render a simple triangle. For example, this tutorial[8]'s code contains 121 lines of JavaScript, 46 being code (not comments or empty lines) to render a simple, non-textured triangle. For

[8] https://www.tutorialspoint.com/webgl/webgl_drawing_a_triangle.htm

this reason, it seems unreasonable to build a system like the one we are describing in raw WebGL. There are many libraires that wrap WebGL code and that help people building 3D interfaces, and THREE.js[9] is a very popular one (56617 stars on github, making it the 35th most starred repository on GitHub as of November 26th, 2019[10]). THREE.js acts as a 3D engine built on WebGL. It provides classes to deal with everything we need:

- the **Renderer** class contains all the WebGL code needed to render a scene on the web page;
- the **Object** class contains all the boilerplate needed to manage the tree structure of the content, it contains a transform (translation and rotation) and it can have children that are other objects;
- the **Scene** class is the root object, it contains all of the objects we want to render and it is passed as argument to the render function;
- the **Geometry** and **BufferGeometry** classes are the classes that hold the vertex buffers, we will discuss it more in the next paragraph;
- the **Material** class is the class that holds the properties used to render geometry (the most important information being the texture), there are many classes derived from Material, and the developer can choose what material they want for their objects;
- the **Mesh** class is the class that links the geometry and the material, it derives the Object class and can thus be added to a scene and rendered.

A snippet of the basic usage of these classes is given in Snippet 1.3.

Geometries. Geometries are the classes that hold the vertices, texture coordinates, normals and faces. THREE.js proposes two classes for handling geometries:

- the **Geometry** class, which is made to be developer friendly and allows easy editing but can suffer from performance issues;
- the **BufferGeometry** class, which is harder to use for a developer, but allows better performance since the developer controls how data is transmitted to the GPU.

1.3.2 Rust

In this section, we explain the specificities of Rust and why it is an adequate language for writing efficient native software safely.

[9] https://threejs.org
[10] https://web.archive.org/web/20191126151645/https://gitstar-ranking.com/mrdoob/three.js

```
// Computes the aspect ratio of the window.
let aspectRatio = window.innerWidth / window.innerHeight;

// Creates a camera and sets its parameters and position.
let camera = new THREE.PerspectiveCamera(70, aspectRatio, 0.01, 10);
camera.position.z = 1;

// Creates the scene that contains our objects.
let scene = new THREE.Scene();

// Creates a geometry (vertices and faces) corresponding to a cube.
let geometry = new THREE.BoxGeometry(0.2, 0.2, 0.2);

// Creates a material that paints the faces depending on their normal.
let material = new THREE.MeshNormalMaterial();

// Creates a mesh that associates the geometry with the material.
let mesh = new THREE.Mesh(geometry, material);

// Adds the mesh to the scene.
scene.add(mesh);

// Creates the renderer and append its canvas to the DOM.
renderer = new THREE.WebGLRenderer({ antialias: true });
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

// Renders the scene with the camera.
renderer.render(scene, camera);
```

Snippet 1.3: A THREE.js *hello world*

Borrow checker

Rust is a system programming language focused on safety. It is made to be efficient (and effectively has performances comparable to C[11] or C++[12]) but with some extra features. C++ users might see it as a language like C++ but that forbids undefined behaviours.[13] The most powerful concept from Rust is *ownership*. Basically, every value has a variable that we call its *owner*. To be able to use a value, you must either be its owner or borrow it. There are two types of borrow, the immutable borrow and the mutable borrow (roughly equivalent to references in C++). The compiler comes with the *borrow checker* which makes sure you only use variables that you are allowed to use. For example, the owner can only use the value if it is not being borrowed, and it is only possible to either mutably

[11] https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust.html
[12] https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust-gpp.html
[13] in Rust, when you need to execute code that might lead to undefined behaviours, you need to put it inside an `unsafe` block. Many operations are not available outside an `unsafe` block (e.g., dereferencing a pointer, or mutating a static variable). The idea is that you can use `unsafe` blocks when you require it, but you should avoid it as much as possible and when you do it, you must be particularly careful.

borrow a value once, or immutably borrow a value many times. At first, the borrow checker seems particularly efficient to detect bugs in concurrent software, but in fact, it is also decisive in non concurrent code. Consider the piece of C++ code in Snippets 1.4 and 1.5.

```
auto vec = std::vector<int> {1, 2, 3};
for (auto value: vec)
    vec.push_back(value);
```

Snippet 1.4: Undefined behaviour with for each syntax

```
auto vec = std::vector<int> {1, 2, 3};
for (auto it = std::begin(vec); it < std::end(vec); it++)
    vec.push_back(*it);
```

Snippet 1.5: Undefined behaviour with iterator syntax

This loop should go endlessly because the vector grows in size as we add elements in the loop. But the most important thing here is that since we add elements to the vector, it will eventually need to be reallocated, and that reallocation will invalidate the iterator, meaning that the following iteration will provoke an undefined behaviour. The equivalent code in Rust is in Snippets 1.6 and 1.7.

```
let mut vec = vec![1, 2, 3];
for value in &vec {
    vec.push(value);
}
```

Snippet 1.6: Rust version of Snippet 1.4

```
let mut vec = vec![1, 2, 3];
let iter = vec.iter();
loop {
    match iter.next() {
        Some(x) => vec.push(x),
        None => break,
    }
}
```

Snippet 1.7: Rust version of Snippet 1.5

What happens is that the iterator needs to borrow the vector. Because it is borrowed, it can no longer be borrowed as mutable since mutating it could invalidate the other borrowers. And effectively, the borrow checker will crash the compiler with the error in Snippet 1.8.

This example is one of the many examples of how powerful the borrow checker is: in Rust code, there can be no dangling reference, and all the segmentation faults coming from them are detected by the compiler. The borrow checker may seem like an enemy to newcomers because it often rejects code that seem correct, but once they get used to it, they understand what is the problem with their code and either fix the problem easily, or realize that the whole architecture is wrong and understand why.

It is probably for those reasons that Rust is the *most loved programming language*

```
error[E0502]: cannot borrow `vec` as mutable because it is also borrowed as
    immutable
 --> src/main.rs:4:9
  |
3 |    for value in &vec {
  |                 ----
  |                 |
  |                 immutable borrow occurs here
  |                 immutable borrow later used here
4 |        vec.push(*value);
  |        ^^^^^^^^^^^^^^^^ mutable borrow occurs here
```

Snippet 1.8: Error given by the compiler on Snippet 1.6

according to the Stack Overflow Developer Survey in 2016, 2017, 2018 and 2019.

Tooling

Moreover, Rust comes with many programs that help developers.

- **rustc**[14] is the Rust compiler. It is comfortable due to the clarity and precise explanations of its error messages.
- **cargo**[15] is the official Rust's project and package manager. It manages compilation, dependencies, documentation, tests, etc.
- **racer**[16], **rls** (Rust Language Server)[17] and **rust-analyzer**[18] are software that manage automatic compilation to display errors in code editors as well as providing semantic code completion.
- **rustfmt**[19] auto formats code.
- **clippy**[20] is a linter that detects unidiomatic code and suggests modifications.

Glium

When we need to perform rendering for 3D content analysis or for evaluation, we use the glium[21] library. Glium has many advantages over using raw OpenGL calls. Its objectives are:

- to be easy to use: it exposes functions that are higher level than raw OpenGL calls, but still low enough level to let the developer free;

[14] https://github.com/rust-lang/rust
[15] https://github.com/rust-lang/cargo
[16] https://github.com/racer-rust/racer
[17] https://github.com/rust-lang/rls
[18] https://github.com/rust-analyzer/rust-analyzer
[19] https://github.com/rust-lang/rustfmt
[20] https://github.com/rust-lang/rust-clippy
[21] https://github.com/glium/glium

- to be safe: debugging OpenGL code can be a nightmare, and glium does its best to use the borrow checker to its advantage to avoid OpenGL bugs;
- to be fast: the binary produced use optimized OpenGL functions calls;
- to be compatible: glium seeks to support the latest versions of OpenGL functions and falls back to older functions if the most recent ones are not supported on the device.

Conclusion

In our work, many tasks will consist in 3D content analysis, reorganization, rendering and evaluation. Many of these tasks require long computations, lasting from hours to entire days. To perform them, we need a programming language that has good performances. In addition, the extra features that Rust provides ease tremendously development, and this is why we use Rust for all tasks that do not require having a web interface.

Chapter 2

Related work

Contents

In this chapter, we review the part of the state of the art on multimedia streaming and interaction that is relevant for this thesis. As discussed in the previous chapter, video and 3D share many similarities and since there is already a very important body of work on video streaming, we start this chapter with a review of this domain with a particular focus on the DASH standard. Then, we proceed with presenting topics related to 3D streaming, including compression and streaming, geometry and texture compromise, and viewpoint dependent streaming. Finally, we end this chapter by reviewing the related work regarding 3D navigation and interfaces.

2.1 Video

Accessing a remote video through the web has been a widely studied problem since the 1990s. The Real-time Transport Protocol (RTP, Schulzrinne et al. [1996]) has been an early attempt to formalize audio and video streaming. The protocol allowed data to be transferred unilaterally from a server to a client, and required the server to handle a separate session for each client.

In the following years, HTTP servers have become ubiquitous, and many industrial actors (Apple, Microsoft, Adobe, etc.) developed HTTP streaming systems to deliver multimedia content over the network. In an effort to bring interoperability between all different actors, the MPEG group launched an initiative, which eventually became a standard known as DASH, Dynamic Adaptive Streaming over HTTP. Using HTTP for multimedia streaming has many advantages over RTP. While RTP is stateful (that is to say, it requires keeping track of every user along the streaming session), HTTP is stateless, and thus more efficient. Furthermore, an HTTP server can easily be replicated at different geographical locations, allowing users to fetch data from the closest server. This type of network architecture is called CDN (Content Delivery Network) and increases the speed of HTTP requests, making HTTP based multimedia streaming more efficient.

2.1.1 DASH: the standard for video streaming

Dynamic Adaptive Streaming over HTTP (DASH), or MPEG-DASH [Stockhammer, 2011; Sodagar, 2011], is now a widely deployed standard for adaptively streaming video on the web [DASH, 2014], made to be simple, scalable and inter-operable. DASH describes guidelines to prepare and structure video content, in order to allow a great adaptability of the streaming without requiring any server side computation. The client should be able to make good decisions on what part of the content to download, only based on an estimation of the network constraints and on the information provided in a descriptive file: the MPD.

DASH structure

All the content structure is described in a Media Presentation Description (MPD) file, written in the XML format. This file has 4 layers: the periods, the adaptation sets, the representations, and the segments. An MPD has a hierarchical structure, meaning it has multiple periods, and each period can have multiple adaptation sets, each adaptation set can have multiple representation, and each representation can have multiple segments.

Periods. Periods are used to delimit content depending on time. It can be used to delimit chapters, or to add advertisements that occur at the beginning, during or at the end of a video.

Adaptation sets. Adaptation sets are used to delimit content according to the format. Each adaptation set has a mime-type, and all the representations and segments that it contains share this mime-type. In videos, most of the time, each period has at least one adaptation set containing the images, and one adaptation set containing the sound. It may also have an adaptation set for subtitles.

Representations. The representation level is the level DASH uses to offer the same content at different levels of quality. For example, an adaptation set containing images has a representation for each available quality (it might be 480p, 720p, 1080p, etc.). This allows a user to choose its representation and change it during the video, but most importantly, since the software is able to estimate its downloading speed based on the time it took to download data in the past, it is able to find the optimal representation, being the highest quality that the client can request without stalling.

Segments. Until this level in the MPD, content has been divided but it is still far from being sufficiently divided to be streamed efficiently. A representation of the images of a chapter of a movie is still a long video, and keeping such a big file is not possible since heavy files prevent streaming adaptability: if the user requests to change the quality of a video, the system would either have to wait until the file is totally downloaded, or cancel the request, making all the progress done unusable.

Segments are used to prevent this issue. They typically encode files that contain two to ten seconds of video, and give the software a greater ability to dynamically adapt to the system. If a user wants to seek somewhere else in the video, only one segment of data is potentially lost, and only one segment of data needs to be downloaded for the playback to resume. The impact of the segment duration has been investigated in many work, including [Sideris et al., 2015; Stohr et al., 2017]. For example, [Stohr et al., 2017] discuss how the segment duration affects the streaming: short segments lower the initial

delay and provide the best stalling quality of experience, but make the total downloading time of the video longer because of overhead.

Content preparation and server

Encoding a video in DASH format consists in partitioning the content into periods, adaptation sets, representations and segments as explained above, and generating a Media Presentation Description file (MPD) which describes this organization. Once the data are prepared, they can simply be hosted on a static HTTP server which does no computation other than serving files when it receives requests. All the intelligence and the decision making is moved to the client side. This is one of the DASH strengths: no powerful server is required, and since static HTTP server are mature and efficient, all DASH clients can benefit from it.

Client side adaptation

A client typically starts by downloading the MPD file, and then proceeds on downloading segments from the different adaptation sets. While the standard describes well how to structure content on the server side, the client may be freely implemented to take into account the specificities of a given application. The most important part of any implementation of a DASH client is called the adaptation logic. This component takes into account a set of parameters, such as network conditions (bandwidth, throughput, for example), buffer states or segments size to derive a decision on which segments should be downloaded next. Most of the industrial actors have their own adaptation logic, and many more have been proposed in the literature. A thorough review is beyond the scope of this state-of-the-art, but examples include [Chiariotti et al., 2016] who formulate the problem in a reinforcement learning framework, [Yadav et al., 2017] who formulate the problem using queuing theory, or [Huang et al., 2019] who use a formulation derived from the knapsack problem.

2.1.2 DASH-SRD

Being now widely adopted in the context of video streaming, DASH has been adapted to various other contexts. DASH-SRD (Spatial Relationship Description, [Niamut et al., 2016]) is a feature that extends the DASH standard to allow streaming only a spatial subpart of a video to a device. It works by encoding a video at multiple resolutions, and tiling the highest resolutions as shown in Figure 2.1. That way, a client can choose to download either the low resolution of the whole video or higher resolutions of a subpart of the video.

For each tile of the video, an adaptation set is declared in the MPD, and a supplemental property is defined in order to give the client information about the tile. This supplemental

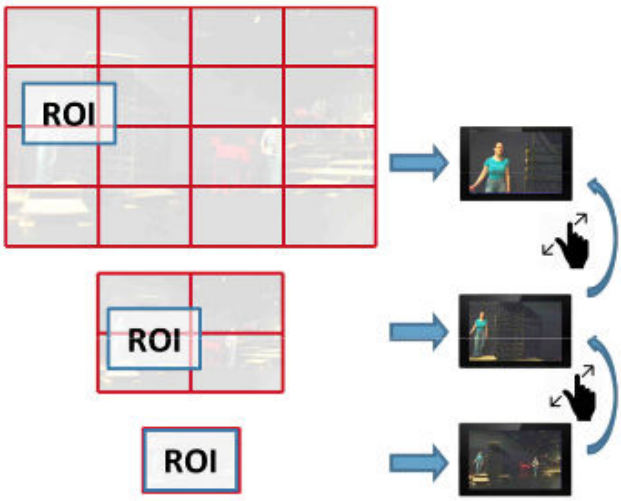


Figure 2.1: DASH-SRD [Niamut et al., 2016]

property contains many elements, but the most important ones are the position (x and y) and the size (width and height) describing the position of the tile in relation to the full video. An example of such a property is given in Snippet 2.1.

```
<Period>
 <AdaptationSet>
  <SupplementalProperty schemeIdUri="urn:mpeg:dash:s rd:2014" value="
      0,0,0,5760,3240,5760,3240"/>
  <Role schemeIdUri="urn:mpeg:dash:role:2011" value= "main"/>
  <Representation id="1" width="3840" height="2160">
   <BaseURL>full.mp4</BaseURL>
  </Representation>
 </AdaptationSet>
 <AdaptationSet>
  <SupplementalProperty schemeIdUri="urn:mpeg:dash:s rd:2014" value="
      0,1920,1080,1920,1080,5760,3240"/>
  <Role schemeIdUri="urn:mpeg:dash:role:2011" value= "supplementary"/>
  <Representation id="2" width="1920" height="1080">
   <BaseURL>part.mp4</BaseURL>
  </Representation>
 </AdaptationSet>
</Period>
```

Snippet 2.1: MPD of a video encoded using DASH-SRD

Essentially, this feature is a way of achieving view-dependent streaming, since the client only displays a part of the video and can avoid downloading content that will not be displayed. While Figure 2.1 illustrates how DASH-SRD can be used in the context of zoomable video streaming, the ideas developed in DASH-SRD have proven to be particularly useful in the context of 360 video streaming (see for example [Ozcinar et al.,

2017]). This is especially interesting in the context of 3D streaming since we have this same pattern of a user viewing only a part of a content.

2.2 3D streaming

In this thesis, we focus on the objective of delivering large, massive 3D scenes over the network. While 3D streaming is not the most popular research field, there has been a special attention around 3D content compression, in particular progressive compression which can be considered a premise for 3D streaming. In the next sections, we review the 3D streaming related work, from 3D compression and structuring to 3D interaction.

2.2.1 Compression and structuring

According to [Maglo et al., 2015], mesh compression can be divided into four categories:

- single-rate mesh compression, seeking to reduce the size of a mesh;
- progressive mesh compression, encoding meshes in many levels of resolution that can be downloaded and rendered one after the other;
- random accessible mesh compression, where different parts of the models can be decoded in an arbitrary order;
- mesh sequence compression, compressing mesh animations.

Since our objective is to stream 3D static scenes, single-rate mesh and mesh sequence compressions are less interesting for us. This section thus focuses on progressive meshes and random accessible mesh compression.

Progressive meshes were introduced in [Hoppe, 1996] and allow a progressive transmission of a mesh by sending a low resolution mesh first, called *base mesh*, and then transmitting detail information that a client can use to increase the resolution. To do so, an algorithm, called *decimation algorithm*, starts from the original full resolution mesh and iteratively removes vertices and faces by merging vertices through the so-called *edge collapse* operation (Figure 2.2).

Every time two vertices are merged, a vertex and two faces are removed from the original mesh, decreasing the model resolution. At the end of this content preparation phase, the mesh has been reorganized into a base mesh and a sequence of partially ordered edge split operations. Thus, a client can start by downloading the base mesh, display it to the user, and keep downloading refinement operations (vertex splits) and display details as time goes by. This process reduces the time a user has to wait before seeing a downloaded 3D object, thus increases the quality of experience.

[Lavoué et al., 2013] develop a dedicated progressive compression algorithm based on iterative decimation, for efficient decoding, in order to be usable on web clients. With

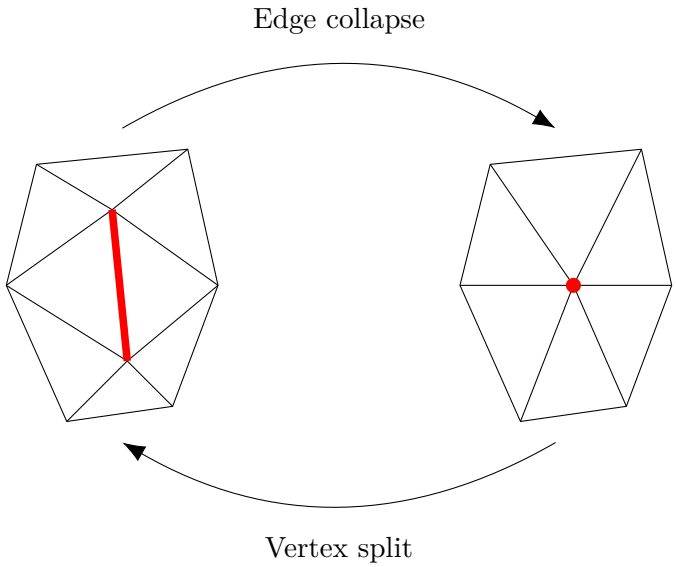


Figure 2.2: Vertex split and edge collapse

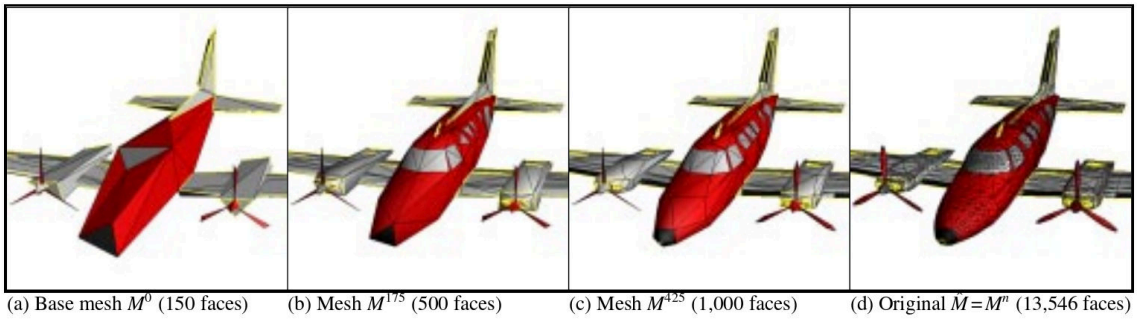


Figure 2.3: Four levels of resolution of a mesh [Hoppe, 1996]

the same objective, [Limper et al., 2013] proposes pop buffer, a progressive compression method based on quantization that allows efficient decoding.

Following these, many approaches use multi triangulation, which creates mesh fragments at different levels of resolution and encodes the dependencies between fragments in a directed acyclic graph. In [Cignoni et al., 2005], the authors propose Nexus: a GPU optimized version of multi triangulation that pushes its performances to make real time rendering possible. It is notably used in 3DHOP (3D Heritage Online Presenter, [Potenziani et al., 2015]), a framework to easily build web interfaces to present 3D objects to users in the context of cultural heritage.

Each of these approaches define its own compression and coding for a single mesh. However, users are often interested in scenes that contain multiple meshes, and the need to structure content emerged.

To answer those issues, the Khronos group proposed a generic format called glTF (GL Transmission Format, [Robinet and Cozzi, 2013]) to handle all types of 3D content rep-

resentations: point clouds, meshes, animated models, etc. glTF is based on a JSON file, which encodes the structure of a scene of 3D objects. It contains a scene graph with cameras, meshes, buffers, materials, textures and animations. Although relevant for compression, transmission and in particular streaming, this standard does not yet consider view-dependent streaming, which is required for large scene remote visualization and which we address in our work.

2.2.2 Viewpoint dependency

3D streaming means that content is downloaded while the user is interacting with the 3D object. In terms of quality of experience, it is desirable that the downloaded content falls into the user's field of view. This means that the progressive compression must encode a spatial information in order to allow the decoder to determine content adapted to its viewpoint. This is typically called *random accessible mesh compression*. [Maglo et al., 2013] is such an example of random accessible progressive mesh compression. [Cheng and Ooi, 2008a] proposes a receiver driven way of achieving viewpoint dependency with progressive mesh: the client starts by downloading the base mesh, and from then is able to estimate the importance of the different vertex splits, in order to choose which ones to download. Doing so drastically reduces the server computational load, since it only has to send data, and improves the scalability of this framework.

In the case of streaming a large 3D scene, view-dependent streaming is fundamental: a user will only be seeing one small portion of the scene at each time, and a system that does not adapt its streaming to the user's point of view is bound to induce a low quality of experience.

A simple way to implement viewpoint dependency is to request the content that is spatially close to the user's camera. This approach, implemented in Second Life and several other NVEs (e.g., [Liang et al., 2011]), only depends on the location of the avatar, not on its viewing direction. It exploits spatial coherence and works well for any continuous movement of the user, including turning. Once the set of objects that are likely to be accessed by the user is determined, the next question is in what order should these objects be retrieved. A simple approach is to retrieve the objects based on distance: the spatial distance from the user's virtual location and rotational distance from the user's view.

More recently, Google integrated Google Earth 3D module into Google Maps (Figure 2.4). Users are now able to go to Google Maps, and click the 3D button which shifts the camera from the aerial view. Even though there are no associated publications to support this assertion, it seems clear that the streaming is view-dependent: low resolution from the center of the point of view gets downloaded first, and higher resolution data gets downloaded for closer objects than for distant ones.

Other approaches use level of details. Level of details have been initially used for

Figure 2.4: Screenshot of the 3D interface of Google Maps

efficient 3D rendering [Lindstrom et al., 1996]. When the change from one level of detail to another is direct, it can create visual discomfort to the user. This is called the *popping effect* and level of details have the advantage of enabling techniques, such as geomorhping [Hoppe, 1998], to transition smoothly from one level of detail to another. Level of details have then been used for 3D streaming. For example, [Guthe and Klein, 2004] propose an out-of-core viewer for remote model visualization based by adapting hierarchical level of details [Erikson et al., 2001] to the context of 3D streaming. Level of details can also be used to perform viewpoint dependant streaming, such as [Meng and Zha, 2003].

2.2.3 Texture streaming

In order to increase the texture rendering speed, a common technique is the *mipmapping* technique. It consists in generating progressively lower resolutions of an initial texture. Lower resolutions of the textures are used for polygons which are far away from the camera, and higher resolutions for polygons closer to the camera. Not only this reduces the time needed to render the polygons, but it can also reduce the aliasing effect. Using these lower resolutions can be especially interesting for streaming. [Marvie and Bouatouch, 2003] proposes the PTM format which encode the mipmap levels of a texture that can be downloaded progressively, so that a lower resolution can be shown to the user while the higher resolutions are being downloaded.

Since 3D data can contain many textures, [Simon et al., 2019] propose a way to stream a set of textures by encoding them into a video. Each texture is segmented into tiles of a fixed size. Those tiles are then ordered to minimize dissimilarities between consecutive tiles, and encoded as a video. By benefiting from the video compression techniques, the authors

are able to reach a better rate-distortion ratio than webp, which is the new standard for texture transmission, and jpeg.

2.2.4 Geometry and textures

As discussed in Chapter 1.1, most 3D scenes consist in two main types of data: geometry and textures. When addressing 3D streaming, one must handle the concurrency between geometry and textures, and the system needs to address this compromise.

Balancing between streaming of geometry and texture data is addressed by [Tian and AlRegib, 2008], [Guo et al., 2017], and [Yang et al., 2004]. Their approaches combine the distortion caused by having lower resolution meshes and textures into a single view independent metric. [Portaneri et al., 2019] also deals with the geometry / texture compromise. This work designs a cost driven framework for 3D data compression, both in terms of geometry and textures. The authors generate an atlas for textures that enables efficient compression and multi-resolution scheme. All four works considered a single mesh, and have constraints on the types of meshes that they are able to compress. Since the 3D scenes we are interested in in our work consist in soups of textured polygons, those constraints are not satisfied and we cannot use those techniques.

2.2.5 Streaming in game engines

In traditional video games, including online games, there is no requirement for 3D data streaming. Video games either come with a physical support (CD, DVD, Blu-Ray) or they require the downloading of the game itself, which includes the 3D data, before letting the user play. However, transferring data from the disk to the memory is already a form of streaming. This is why optimized engines for video games use techniques that are reused for streaming such as level of details, to reduce the details of objects far away for the point of view and save the resources to enhance the level of detail of closer objects.

Some other online games, such as Second Life[1], rely on user generated data, and thus are forced to send data from users to others. In such scenarios, 3D streaming is appropriate and this is why the idea of streaming 3D content for video games has been investigated. For example, [Li et al., 2011] proposes an online game engine based on geometry streaming, that addresses the challenge of streaming 3D content at the same time as synchronization of the different players.

2.2.6 NVE streaming frameworks

An example of NVE streaming framework is 3D Tiles [Schilling et al., 2016], which is a specification for visualizing massive 3D geospatial data developed by Cesium and built on

[1] https://secondlife.com

top of glTF. Their main goal is to display 3D objects on top of regular maps, and their visualization consists in a top-down view, whereas we seek to let users freely navigate in our scenes, whether it be flying over the scene or moving along the roads.

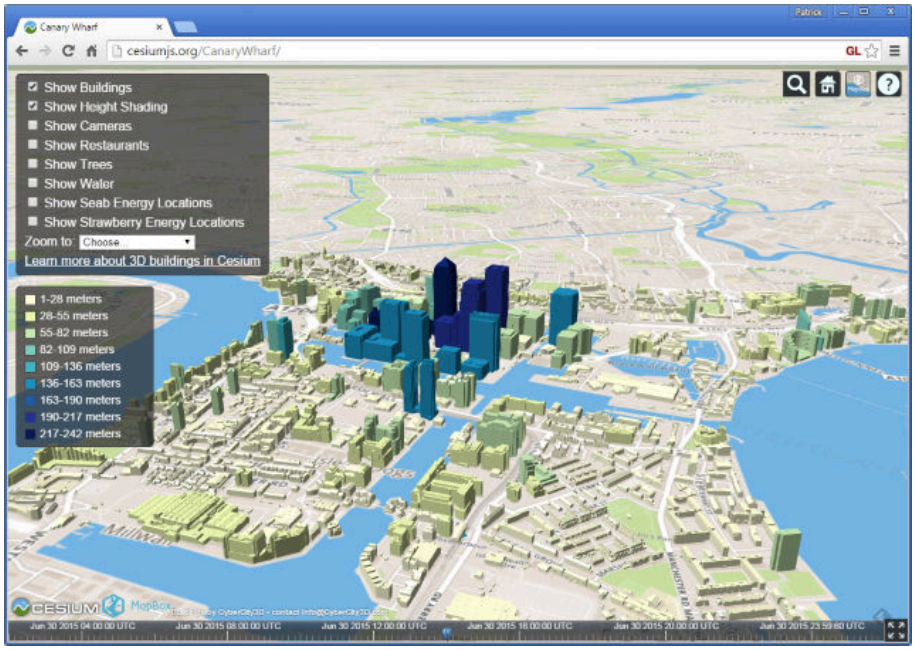


Figure 2.5: Screenshot of 3D Tiles interface [Schilling et al., 2016]

3D Tiles, as its name suggests, is based on a spacial partitionning of the scene. It started with a regular octree, but has then been improved to a k-d tree (see Figure 2.6).

In 2019, 3D Tiles streaming system was improved by preloading the data at the camera's next position when known in advance (with ideas that are similar to those we discuss and implement in Chapter 3, published in 2016) and by ordering tile requests depending on the user's position (with ideas that are similar to those we discuss and implement in Chapter 4, published in 2018a).

[Zampoglou et al., 2018] is another example of a streaming framework: it is the first paper that proposes to use DASH to stream 3D content. In their work, the authors describe a system that allows users to access 3D content at multiple resolutions. They organize the content, following DASH terminology, into periods, adaptation sets, representations and segments. Their first adaptation set codes the tree structure of the scene graph. Each further adaptation set contains both geometry and texture information and is available at different resolutions defined in a corresponding representation. To avoid requests that would take too long and thus introduce latency, the representations are split into segments. The authors discuss the optimal number of polygons that should be stored in a single segment. On the one hand, using segments containing very few faces will induce many HTTP requests from the client, and will lead to poor streaming efficiency. On the other hand, if segments contain too many faces, the time to load the segment is long and the

(a) With regular octree (depth 4) (b) With k-d tree (depth 6)

Figure 2.6: Spatial partitionning used in 3D Tiles

system loses adaptability. Their approach works well for several objects, but does not handle view-dependent streaming, which is desirable in the use case of large NVEs.

2.3 3D bookmarks and navigation aids

One of the uses for 3D streaming is to allow users interacting with the content while it is being downloaded. However, devising an ergonomic technique for browsing 3D environments through a 2D interface is difficult. Controlling the viewpoint in 3D (6 DOFs) with 2D devices is not only inherently challenging but also strongly task-dependent. In their review, [Jankowski and Hachet, 2015] distinguish between several types of camera movements: general movements for exploration (e.g., navigation with no explicit target), targeted movements (e.g., searching and/or examining a model in detail), specified trajectory (e.g., a cinematographic camera path). For each type of movement, specialized 3D interaction techniques can be designed. In most cases, rotating, panning, and zooming movements are required, and users are consequently forced to switch back and forth among several navigation modes, leading to interactions that are too complicated overall for a layperson. Navigation aids and smart widgets are required and subject to research efforts both in 3D companies (see `sketchfab.com`, `cl3ver.com` among others) and in academia, as reported below.

Translating and rotating the camera can be simply specified by a *lookat* point. This is often known as point-of-interest (POI) movement (or *go-to*, *fly-to* interactions) [Mackinlay et al., 1990]. Given such a point, the camera automatically moves from its current

position to a new position that looks at the POI. One key issue of these techniques is to correctly orient the camera at destination. In Unicam [Zeleznik et al., 1997], the so-called click-to-focus strategy automatically chooses the destination viewpoint depending on 3D orientations around the contact point. The more recent Drag'n Go interaction [Moerman et al., 2012] also hits a destination point while offering control on speed and position along the camera path. This 3D interaction is designed in the screen space (it is typically a mouse-based camera control), where cursor's movements are mapped to camera movements following the same direction as the on-screen optical-flow.

Figure 2.7: Screenshot of the drag'n go interface [Moerman et al., 2012] (the percentage widget is for illustration)

Some 3D browsers provide a viewpoint menu offering a choice of viewpoints [Todd, 2004; Burtnyk et al., 2006]. Authors of 3D scenes can place several viewpoints (typically for each POI) in order to allow easy navigation for users, who can then easily navigate from viewpoint to viewpoint just by selecting a menu item. Such viewpoints can be either static, or dynamically adapted: [Jankowski and Decker, 2012] report that users clearly prefer navigating in 3D using a menu with animated viewpoints than with static ones.

Figure 2.8: Screenshot of an interface with menu for navigation [Burtnyk et al., 2006]

Early 3D VRML environments [Rezzonico and Thalmann, 1996] offer 3D bookmarks with animated transitions between bookmarked views. These transitions prevent disorientation since users see how they got there. Hyperlinks can also ease rapid movements between distant viewpoints and naturally support non-linear and non-continuous access to 3D content. Navigating with 3D hyperlinks is faster due to the instant motion, but can cause disorientation, as shown by the work of [Ruddle et al., 2000]. [Eno et al., 2010] examine explicit landmark links as well as implicit avatar-chosen links in Second Life. These authors point out that linking is appreciated by users and that easing linking would likely result in a richer user experience. [Jankowski and Decker, 2012] developed the Dual-Mode User Interface (DMUI) that coordinates and links hypertext to 3D graphics in order to access information in a 3D space.

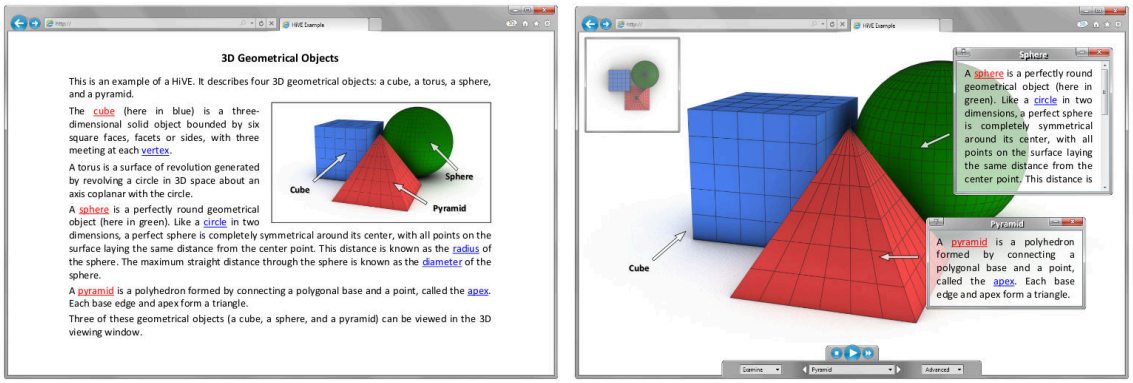


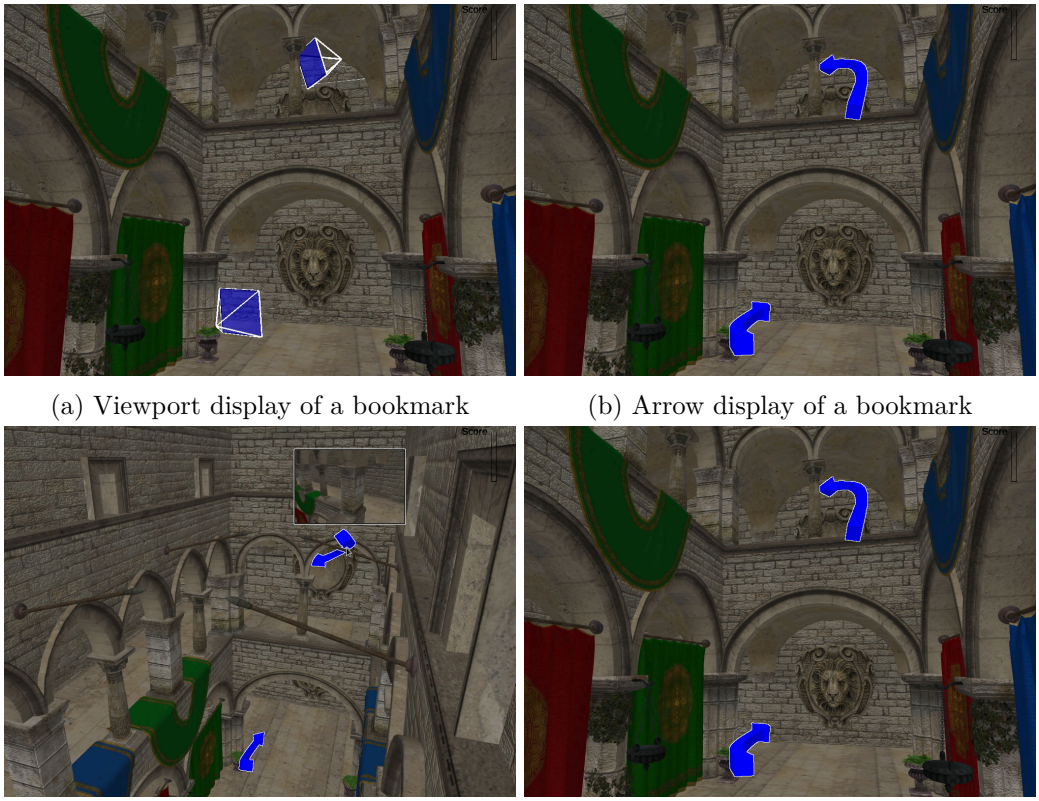
Figure 2.9: The two modes of DMUI [Jankowski and Decker, 2012]

The use of in-scene 3D navigation widgets can also facilitate 3D navigation tasks. [Chittaro and Venkataraman, 2006] propose and evaluate 2D and 3D maps as navigation aids for complex virtual buildings and find that the 2D navigation aid outperforms the 3D one for searching tasks. The ViewCube widget [Khan et al., 2008] serves as a proxy for the 3D scene and offers viewpoint switching between 26 views while clearly indicating associated 3D orientations. Interactive 3D arrows that point to objects of interest have also been proposed as navigation aids in [Chittaro and Burigat, 2004; Burigat and Chittaro, 2007]: when clicked, the arrows transfer the viewpoint to the destination through a simulated walk or a faster flight.

Chapter 3

Bookmarks, navigation and streaming

Contents



(a) Viewport display of a bookmark

(b) Arrow display of a bookmark

(c) A preview is shown when the mouse hovers a bookmark

(d) A coin is hidden behind the curtain

Figure 3.1: 3D bookmarks propose to move to a new viewpoint; when the user clicks on the bookmark, his viewpoint moves to the indicated viewpoint.

In this chapter, we present our first contribution: an analysis of the impact of bookmarks on navigation and streaming.

We implement a 3D navigation interface where the keyboard translates the camera and the mouse rotates it. We augment this interface with 3D bookmarks. When the user's cursor hovers a bookmark, a preview of the point of view is displayed to the user, and when the user clicks on a bookmark, the camera smoothly moves from its current position to the bookmarked point of view. We conduct a within-subject user-study on 51 participants, where each user starts with a tutorial to get used to the 3D navigation controls, and then tries successively to perform a task with and without bookmarks. We show that not only the presence of bookmarks causes a faster task completion, but also that it allows users to see a larger part of the scene during the same time span.

However, in a streaming scenario, this phenomenon leads to higher network requirements to maintain the same quality of service. In the last part of this chapter, we simulate a streaming setup and we show that knowing the positions of the bookmarks beforehand allows us to precompute information that we can reuse during streaming to compensate for the harm caused by the faster navigation with bookmarks.

3.1 Introduction

Navigating in NVE with a large virtual space (most times through a 2D interface) is sometimes cumbersome. In particular, a user may have difficulties reaching the right place to find information. The content provider of the NVE may want to highlight certain interesting features for the users to view and experience, such as a vantage point in a city, an excavation at an archaeological site, or an exhibit in a museum. To allow users to easily find these interesting locations within the NVE, *3D bookmarks* or *bookmarks* for short, can be provided. A bookmark is simply a 3D virtual camera (with position and camera parameters) predefined by the content provider, and can be presented to users in different ways, including as a text link (URL), a thumbnail image, or a 3D object embedded within the NVE itself.

When users click on a bookmark, NVEs commonly provide a "fly-to" animation to transit the camera from the current viewpoint to the destination [Mackinlay et al., 1990; Rezzonico and Thalmann, 1996] to help orient the users within the 3D space. Clicking on a bookmark to fly to another viewpoint leads to reduced data locality. The 3D content at the bookmarked destination viewpoint may overlap less with the current viewpoint. In the worst case, the 3D objects corresponding to the current and destination viewpoints can be completely disjoint. Such movement to a bookmark may lead to a *discovery latency* [Varvello et al., 2011], in which users have to wait for the 3D content for the new viewpoint to be loaded and displayed. An analogy for this situation, in the context of video streaming, is seeking into a segment of video that has not been prefetched yet.

In this chapter, we explore the impact of bookmarks on NVE navigation and streaming, and make several contributions. First, we conducted a crowdsourcing experiment where 51 participants navigated in 3 virtual scenes to complete a task. This experiment serves two purposes: (i) it validates our intuition that bookmarks significantly reduce the number of interactions and navigation time (in average the time needed to complete the task for users with bookmarks is half the time for users without bookmarks); (ii) it produces a set of user interaction traces that we use for subsequent simulation experiments. Second, we quantified the effect of bookmarking on prefetching and visual quality in our experiments. We showed that, without prefetching, the number of correctly rendered pixels right after clicking on bookmarks can drop up to 10% on average. If we prefetch the 3D content from the bookmarks according to the probability of access, we do not limit this drop by more than 5%. Finally, we proposed a method to improve the visual quality after clicking on bookmarks, by exploiting the fact that the visible faces at the bookmark can be precomputed, and by fetching the visible faces only after a bookmark is clicked. We showed that, if the fetching is done during the 1 or 2 seconds of the "fly-to" camera movement from the current viewpoint to the bookmarked viewpoint, it suffices to increase the number of correctly rendered pixels to more than 20%, without wasting bandwidth on

prefetching. Our key message is that, *in addition to easing navigation, bookmarking allows precomputation of visible faces and can significantly reduce interaction latency, without resorting to prefetching*, which may waste bandwidth by prefetching 3D data that will not be needed.

The rest of the chapter consists of the following sections. Section 3.2 describes the 3D bookmarks that we use in our work, along with our experiments to validate the usefulness of bookmarking. Section 3.3 describes the streaming and prefetching mechanisms that we used to simulate our experiments as well as our main findings. Finally, we conclude in Section 3.4.

3.2 Impact of 3D bookmarks on navigation

We now describe an experiment that we conducted on 51 participants, with two goals in mind. First, we want to measure the impact of 3D bookmarks on navigation within an NVE. Second, we want to collect traces from the users so that we can replay them for reproducible experiments for comparing streaming strategies in Section 3.3.

3.2.1 Our NVE

To ease the deployment of our experiments to users in distributed locations on a crowdsourcing platform, we implement a simple web-based NVE client using THREE.js[1]. The NVE server is implemented with node.js[2]. The NVE server streams a 3D scene to the client; the client renders the scene as the 3D content is received.

The user can navigate within the NVE in the following way; the camera can be translated using the arrow keys along four directions: forward, backward, to the left, and to the right. Alternatively, the keys W, A, S and D can also be used for the same actions. This choice was inspired by 3D video games, which often use these keys in conjunction with the mouse to move an avatar. The virtual camera can rotate in four different directions using the keys I, K, J and L. The user can also rotate the camera by dragging the mouse in the desired direction. Finally, following the UI of popular 3D games, we also give users the possibility to lock their pointer and use their mouse as a virtual camera. The mouse movement controls the camera rotation. The user can always choose to lock the pointer, or unlock it using the escape key. The interface also includes a button to reset the camera back to the starting position in the scene.

[1] http://threejs.org
[2] http://nodejs.org

3.2.2 3D bookmarks

Our NVE supports 3D bookmarks. A 3D bookmark, or bookmark for short, is simply a fixed camera location (in 3D space), a view direction, and a focal. Bookmarks visible from the user's current viewpoint are shown as 3D objects in the scene. Figure 3.1 depicts some bookmarks from our NVE.

The user can click on a bookmark object to automatically move and align its viewpoint to that of the bookmark. The movement follows a Hermite curve joining the current viewpoint to the viewpoint of the bookmark. The tangent of the curve is the view direction. The user can hover the mouse pointer over a bookmark object to see a thumbnail view of the 3D scene as seen from the bookmark. (Figure 3.1, bottom left).

In our work, we consider two different possibilities for displaying bookmarks: viewports (Figure 3.1 top left) and arrows (Figure 3.1 top right). A viewport is displayed as a pyramid where the top corresponds to the optical center of its viewpoint and the base corresponds to its image plane. The arrows are view dependent. The bottom of the arrow turns towards the current position, to better visualize the relative position of the bookmark.

Bookmarks allow the user to achieve a large movement within the 3D environment using a single action (a mouse click). Since bookmarks are part of the scene, they are visible only when not hidden by other objects from the scene. We chose size and colors that are salient enough to be easily seen, but not too large to limit the occlusion of regions within the scene. When reaching the bookmark, the corresponding arrow or viewport is not visible anymore, and subsequently will appear in a different color, to indicate that it has been clicked (similar to web links).

3.2.3 User study

We now describe in details our experimental setup and the user study that we conducted on 3D navigation.

Models

We use four 3D scenes (one for the tutorial and three for the actual experiments) which represent recreated scenes from a famous video game. Those models are light (a few thousand of triangles per model) and are sent before the experiment starts. We keep the models small so that users can perform the task with acceptable latency from any country using a decent internet connection. Our NVE does not stream the 3D content for these experiments, in order to avoid unreliable conditions caused by the network bandwidth variation, which might affect how the users interact.

Task design

Since we are interested in studying how efficiently users navigate in the 3D scene, we ask our participants to complete a task which forces them to visit, at least partially, various regions in the scene. To this end, we hide a set of 8 coins on the scene: participants are asked to collect the coins by clicking on them. In order to avoid any bias due to the coins position, we predefined 50 possible coin locations per scene, and randomly select 8 out of these 50 positions each time a new participant starts the experiment.

Experiment

Participants are first presented with an initial screen to collect some preliminary information: age, gender, the last time they played 3D video games, and self-rated 3D gaming skills. We ask those questions because we believe that someone who is used to playing 3D video games should browse the scene more easily, and thus, may not need to use our bookmarks.

Then, the participants go through a tutorial to learn how the UI works, and how to complete the task. The different interactions (keyboard navigation, mouse navigation, bookmarks interaction) are progressively introduced to participants, and the tutorial ends once the participant completes an easy version of the task. The tutorial is always performed on the same scene.

Then, each participant has to complete the task three times. Each task is performed on a different scene, with a different interface. Three interfaces are used. A NoBM interface lets the participant navigates without any bookmarks. The other two interfaces allow a participant to move using bookmarks displayed as viewports (denoted as VP) and arrows (denoted as Ar) respectively.

The coins are chosen randomly, based on the coin configurations that were used by previous participants: if another participant has done an experiment with a certain set of coins, on a certain scene, with a certain type of bookmarks, the current participant will do the experiment with the same set of coins, on the same scene, but with a different type of bookmarks. This policy allows us to limit the bias that could be caused by coin locations.

Once a participant has found all coins, a button is shown on the interface to let the participant move to the next step. Alternatively, this button may appear one minute after the sixth coin was found. This means that a user is authorized to move on without completing the task, in order to avoid potential frustration caused by not finding the remaining two coins.

After completing the three tasks, the participants have to answer a set of questions about their experience with the bookmarks (we refer to the bookmarks as *recommendations* in the experiments). Table 3.1 shows the list of questions.

Participants. The participants were recruited on microworkers.com, a crowdsourcing

	Questions	Answers
1	What was the difficulty level WITHOUT recommendation?	3.04 / 5 ±0.31
2	What was the difficulty level WITH recommendation?	2.15 / 5 ±0.30
3	Did the recommendations help you to find the coins?	42 Yes, 5 No
4	Did the recommendations help you to browse the scene?	49 Yes, 2 No
5	Do you think recommendations can be helpful?	49 Yes, 2 No
6	Which recommendation style do you prefer and why?	32 Ar, 7 VP
7	Did you enjoy this?	36 Yes, 3 No

Table 3.1: List of questions in the questionnaire and summary of answers. Questions 1 and 2 have a 99% confidence interval.

website. There were 51 participants (36 men and 15 women), who are in average 30.44 years old.

3.2.4 Experimental results

We now present the results from our user study, focusing on whether bookmarks help users navigating the 3D scene.

Questionnaire

We had 51 responses to the questionnaire. The answers are summarized in Table 3.1. Note that not all questions were answered by all participants.

The participants seem to find the task to be of average difficulty (3.04/5) when they have no bookmarks to help their navigation. They judge the task to be easier in average (2.15/5) with bookmarks, which indicates that bookmarks ease the completion of the task.

Almost all users (49 out of 51) think the bookmarks are useful for browsing the scene, and most users (42 out of 51) think bookmarks are also useful to complete the given task. This is slightly in contradiction with our setup; even if coins may appear in some bookmarked viewpoints (which is normal since the viewpoints have been chosen to get the most complete coverage of the scene), most of the time no coin is visible in a given bookmark, and there are always coins that are invisible from all bookmarks.

The strongest result is that almost all users (49 out of 51) find bookmarks to be helpful. In addition, users seem to have a preference for Ar against VP (32 against 7).

Analysis of interactions

Table 3.2 shows basic statistics on task completion given the type of bookmarks that were provided to the participants.

First, we can see that without bookmarks, only a little bit more than a third of the users are able to complete the task, i.e. find all 8 coins. In average, these users find just

BM type	#Exp	Mean # coins	# completed	Mean time
NoBM	51	7.08	18	4 min 16 s
Ar	51	7.39	27	2 min 33 s
VP	51	7.51	30	2 min 16 s

Table 3.2: Analysis of the sessions length and users success by type of bookmarks

above 7 coins, and spend 4 minutes and 16 seconds to do it.

Interestingly, and regardless of the bookmark type, users who have bookmarks complete the task more than half of the time, and spend in average significantly less time to complete the task: 2 minutes and 16 seconds using VP and 2 minutes and 33 seconds using Ar. Although VP seem to help users a little bit more in completing the task than Ar, the performance difference between both types of bookmarks is not significant enough to conclude on which type of bookmarks is best.

The difference between an interface with bookmarks and without bookmarks, however, is very clear. Users tend to complete the task more efficiently using bookmarks: more users actually finish the task, and it takes them half the time to do so. We computed 99% confidence intervals on the results introduced in Table 3.2. We found that the difference in mean number of coins collected with and without bookmarks is not high enough to be statistically significant: we would need more experiments to reach the significance. The mean time spent on the task however is statistically significant.

BM type	Total distance	Distance to a bookmark	Ratio
NoBM	610.80	0	0%
Ar	586.30	369.77	63%
VP	546.96	332.72	61 %

Table 3.3: Analysis of the length of the paths by type of bookmarks

Table 3.3 presents the length of the paths traveled by users in the scenes. Although users tend to spend less time on the tasks when they do not have bookmarks, they travel pretty much the same distance as without bookmarks. As a consequence, they visit the scene faster in average with bookmarks, than without bookmarks. The table shows that this higher speed is due to the bookmarks, as more than 60% of the distance traveled by users with bookmarks happens when users click on bookmarks and fly to the destination.

Discussion

In the previous paragraphs, we have shown how bookmarks are well perceived by users (looking at the questionnaire answers). We also showed that users tend to be more efficient in completing the task when they have bookmarks than when they do not.

We can say that bookmarks have a positive impact on navigation within the 3D scene, but since users move, on average, twice as fast, it might have a negative impact on the streaming of objects to the client.

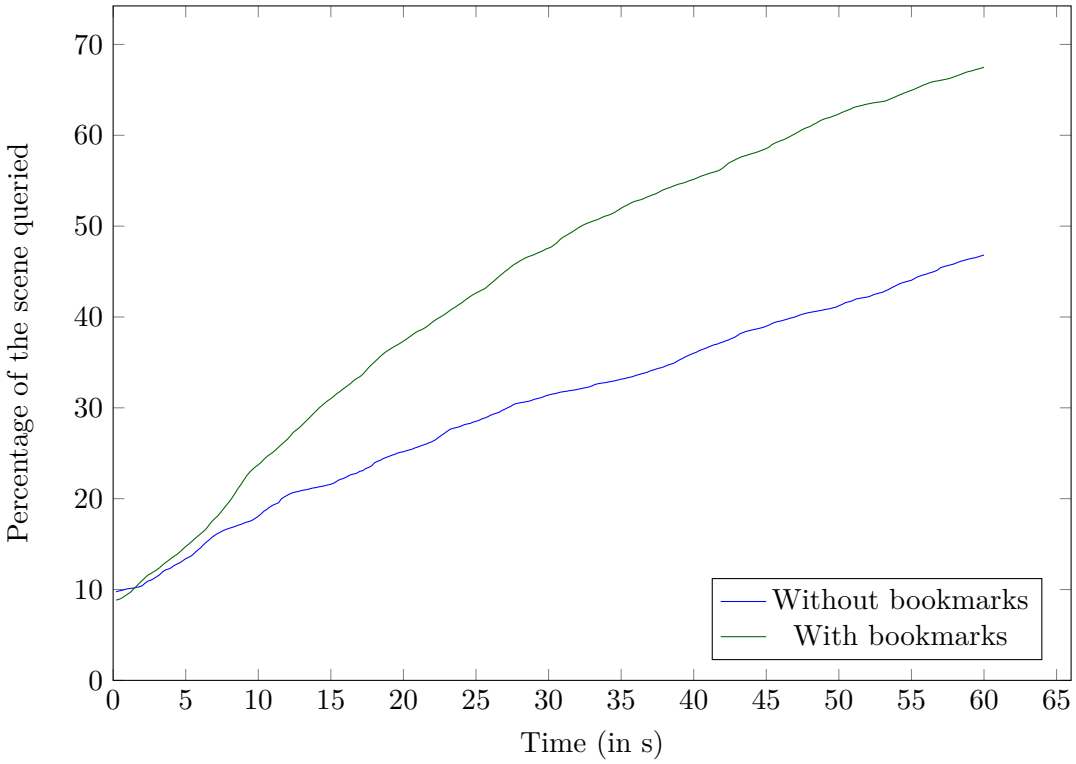


Figure 3.2: Comparison of the triangles queried after a certain time

Figure 3.2 shows a CDF of the percentage of 3D mesh triangles in the scene that have been queried by users after a certain time. We plotted this same curve for users with and without bookmarks. As expected, the fact that the users can browse the scene significantly quicker with bookmarks reflects on the demand on the 3D content. Users need more triangles more quickly, which either leads to more demand on network bandwidth, or if the bandwidth is kept constant, leads to fewer objects being displayed. In the next section, we introduce experiments based on our user study traces that show how the rendering is affected by the presence of bookmarks and how to improve it.

3.3 Impact of 3D bookmarks on streaming

3.3.1 3D model streaming

In this section, we describe our implementation of a 3D model streaming policy in our simulation. A summary of the streaming policies we designed is given in Table 3.5. Note that the policy is different from the one we used for the crowdsourcing experiments. Recall

that in the crowdsourcing experiments, we load all the 3D content before the participants begin to navigate to remove bias due to different network conditions. Here, we implemented a streaming version, which we expect an actual NVE will use.

The 3D content we used are textured mesh — coded in `obj` file format. As such, the data we used in our experiments are made of several components. The geometry consists of (i) a list of vertices and (ii) a list of faces, and the texture consists of (i) a list of materials, (ii) a list of texture coordinates, and (iii) a set of texture images. In the crowdsourcing experiment, we keep the model small since the goal is to study the user interaction. To increase the size of the model, while keeping the same 3D scene, we subdivide each triangle three times, successively, thereby multiplying the total number of triangles in the scene by 64. We do this to simulate a reasonable use case with large 3D scenes. Table 3.4 shows that material and texture amount at most for 3.6% of the geometry, which justifies this choice.

When a client starts loading the web page containing the 3D model, the server first sends the list of materials and the texture files. Then, the server periodically sends a fixed size chunk that indifferently encapsulates vertices, texture coordinates, or faces. A *vertex* is coded with three floats and an integer (x, y, and z coordinates and the index of the vertex), a *texture coordinate* with two floats and an integer (the x and y coordinates on the image and the index of the texture coordinate), and a face with eight integers (the index of each vertex, the index of each texture coordinate, the index of the face and the number of the corresponding material). Consequently, given the JavaScript implementation of integers and floats, we approximate each vertex and each texture coordinate to take up 32 bytes, and each face takes up 96 bytes.

Scene	**Material**	**Images**	**Geometry**
Scene 1	8 KB	72 KB	8.48 MB
Scene 2	302 KB	8 KB	8.54 MB
Scene 3	16 KB	92 KB	5.85 MB

Table 3.4: Respective sizes of materials, textures (images) and geometries for the three scenes used in the user study.

During playback, the client periodically (every 200 ms in our implementation) sends to the server its current position and camera orientation. The server computes a sorted list of relevant faces: first the server performs frustum culling to compute the list of faces that intersect with the client's viewing frustum. Then, it performs backface culling to discard the faces whose normals point towards the same direction as the client's camera orientation. The server then sorts the filtered faces according to their distance to the camera. Finally, the server incrementally fills in chunks with these ordered faces. If a face depends on a vertex or a texture coordinate that has not yet been sent, the vertex or

the texture coordinate is added to the chunk as well. When the chunk is full, the server sends it. Both client and server algorithms are detailed in algorithms 3 and 4. The chunk size is set according to the bandwidth limit of the server. Note that the server may send faces that are occluded and not visible to the client, since determining visibility requires additional computation.

Algorithm 3: Client slide algorithm

```
while streaming is not finished do
    Receive chunk from the server;
    Add the faces from the chunk to the model;
    Update the camera (by 200ms);
    Compute the rendering and evaluate the quality;
    Send the position of the camera to the server;
end
```

Algorithm 4: Server side algorithm

```
while streaming is not finished do
    Receive position of the camera from the client;
    Compute the list of triangles to send and sort them;
    Send a chunk of a certain amount of triangles;
end
```

In the following, we shall denote this streaming policy culling; in Figures 3.6 and 3.7 streaming using culling only is denoted C-only.

3.3.2 3D bookmarks

We have seen (Figure 3.2) that navigation with bookmarks is more demanding on the bandwidth. We want to exploit bookmarks to improve the user's quality of experience. For this purpose, we propose two streaming policies based on offline computation of the relevance of 3D content to bookmarked viewpoints.

Visibility determination for 3D bookmarks

A bookmarked viewpoint is more likely to be accessed, compared to other arbitrary viewpoint in the 3D scene. We exploit this fact to perform some precomputation on the 3D content visible from the bookmarked viewpoint.

Recall that culling does not consider occlusion of the faces. Furthermore, it prioritizes the faces according to distance from the camera, and does not consider the actual contribution of the faces to the rendered 2D images. Ideally, we should prioritize the faces that occupy a bigger area in the 2D rendered images. Computing this, however, requires rendering the

scene at the server, and measuring the area of each face. It is not scalable to compute this for every viewpoint requested by the client.

However, we can prerender the bookmarked viewpoints, since the number of bookmarks is limited, their viewpoints are known in advance, and they are likely to be accessed. For each bookmark, we render the scene offline, using a single color per triangle. Once rendered, we scan the output image to find the visible triangles (based on the color) and sort them by decreasing projected area. This technique is also used by [Cheng and Ooi, 2008b]. Thus, when the user clicks on a 3D bookmark, this precomputed list of faces is used by the server, and only visible faces are sent in decreasing order of contributions to the rendered image.

For the three scenes that we used in the experiment, we can reduce the number of triangles sent by 60% (over all bookmarks). This reduction is as high as 85.7% for one particular bookmark (from 26,886 culled triangles to 3,853 culled and visible triangles).

To illustrate the impact of sorting by projected area of faces, Figure 3.3 shows the quality improvement gained by sending the precomputed visible triangles prioritized by projected areas, compared to using culling only prioritized by distance. The curve shows the average quality over all bookmarks over all scenes, for a given number of triangles received. The quality is measured by the ratio of correctly rendered pixels, comparing the fully and correctly rendered image (when all 3D content is available) and the rendered image (when content is partially available). We sample one pixel every 100 rows and every 100 columns to compute this value. The figure shows that, to obtain 90% of correctly displayed samples, we require 1904 triangles instead of 5752 triangles, about 1/3 savings.

In what follows, we will refer to this streaming policy as visible.

Prefetching by predicting the next bookmark clicked

We can now use the precomputed, visibility-based streaming of 3D content for the bookmarks to reduce the amount of traffic needed. Next, we propose to prefetch the 3D content from the bookmarks. Any efficient prefetching policy needs to accurately predict users' actions.

As shown, users tend to visit the bookmarked viewpoints more often than others, except the initial viewpoint. It is thus natural to try to prefetch the 3D content of the bookmarks.

Figure 3.4 shows the probability of visiting a bookmark (vertical axis) given that another bookmark has been visited (horizontal axis). This figure shows that users tend to follow similar paths when consuming bookmarks. Thus, we hypothesize that prefetching along those paths would lead to better image quality and lower discovery latency.

The policy used is the following. We divide each chunk sent by the server into two parts. The first part is used to fetch the content from the current viewpoint, using the culling streaming policy. The second part is used to prefetch content from the bookmarks, according to their likelihood of being clicked next. We use the probabilities displayed

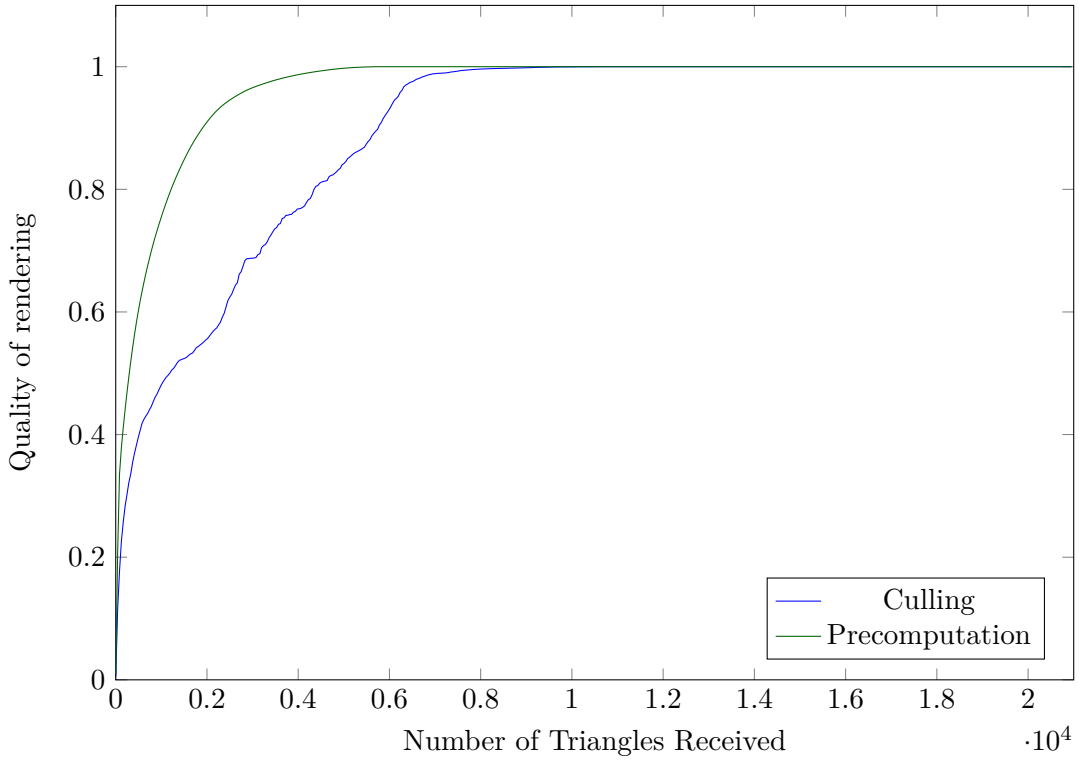


Figure 3.3: Comparison of rendered image quality (average on all bookmarks and starting position): the triangles are sorted offline (green curve), or sorted online by distance to the viewpoint (blue curve).

in Figure 3.4 to determine the size of each part. Each bookmark B has a probability $p(B|B_{prev})$ of being clicked next, considering that B_{prev} was the last clicked bookmark. We assign to each bookmark a certain portion of the chunk to prefetch the corresponding data proportionally to the probability of it being clicked. We use the visible policy to determine which data should be sent for a bookmark.

We denote this combination as V-PP, for Prefetching based on Prediction using visible policy.

Fetching destination bookmark

An alternate method to benefit from the precomputing visible triangles at the bookmark, is to fetch 3D content during the "fly-to" transition to reach the destination. Indeed, as specified in Section 3.2, moving to a bookmarked viewpoint is not instantaneous, but rather takes a small amount of time to smoothly move the user camera from its initial position towards the bookmark. This transition usually takes from 1 to 2 seconds, depending on how far the current user camera position is from the bookmark.

When the user clicks on the bookmark, the client fetches the visible vertices from the destination viewpoint, with all the available bandwidth. So, during the transition time,

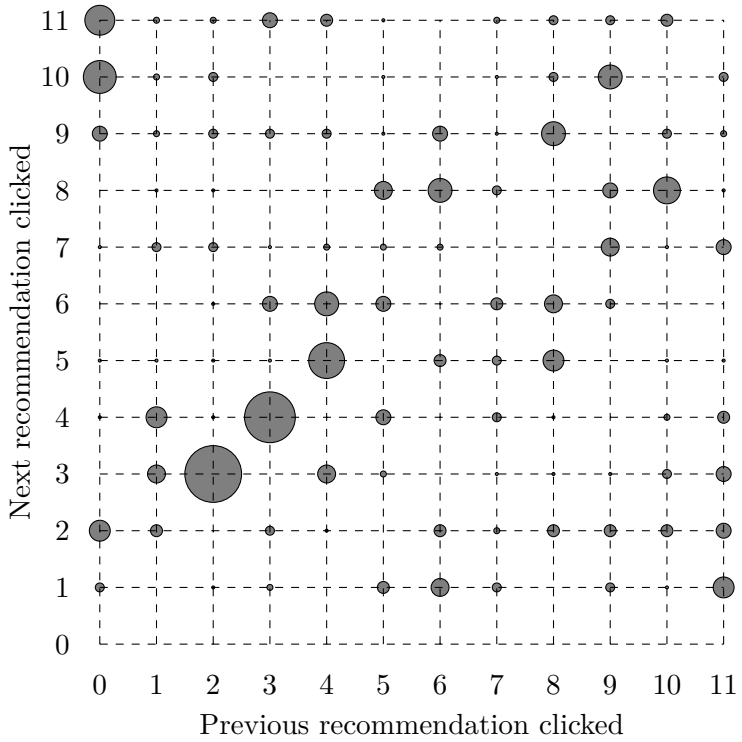


Figure 3.4: Probability distribution of 'next clicked bookmark' for Scene 1 (computed from the 33 users with bookmarks). Numbering corresponds to 0 for initial viewport and 11 bookmarks; the size of the disk at (i, j) is proportional to the probability of clicking bookmark j after i.

Figure 3.5: Example of how a chunk can be divided into fetching what is needed to display the current viewport (culling), and prefetching three recommendations according to their probability of being visited next.

the server no longer does culling, but the whole chunk is used for fetching following visible policy.

The immediate drawback of this policy is that on the way to the bookmark, the user perception of the scene will be degraded because of the lack of data for the viewpoints in transition. On the bright side, no time is lost to prefetch bookmarks that will never be consumed, because we fetch only when we are sure that the user has clicked on a bookmark. This way, when the user is not clicking on bookmarks, we can use the entire bandwidth for the current viewpoint and get as many triangles as possible to improve the current viewpoint. We call this method V-FD, since we are Fetching the 3D data from the Destination using visible policy.

	Visible	V-FD	V-PP	V-PP+FD
Frustum culling	✓	✓	✓	✓
Fetch destination	✗	✓	✗	✓
Prefetch predicted	✗	✗	✓	✓

Table 3.5: Summary of the streaming policies

3.3.3 Comparing streaming policies

In order to determine which policy to use, we replay the traces from the user study while simulating different streaming policies. The first point we are interested in is which streaming policy leads to the lower discovery latency and better image quality for the user: culling (no prefetching), V-PP (prefetching based on probability of accessing bookmarks), or V-FD (no prefetching, but fetch the destination during fly-to transition) or combining both V-PP and V-FD (V-PP+FD).

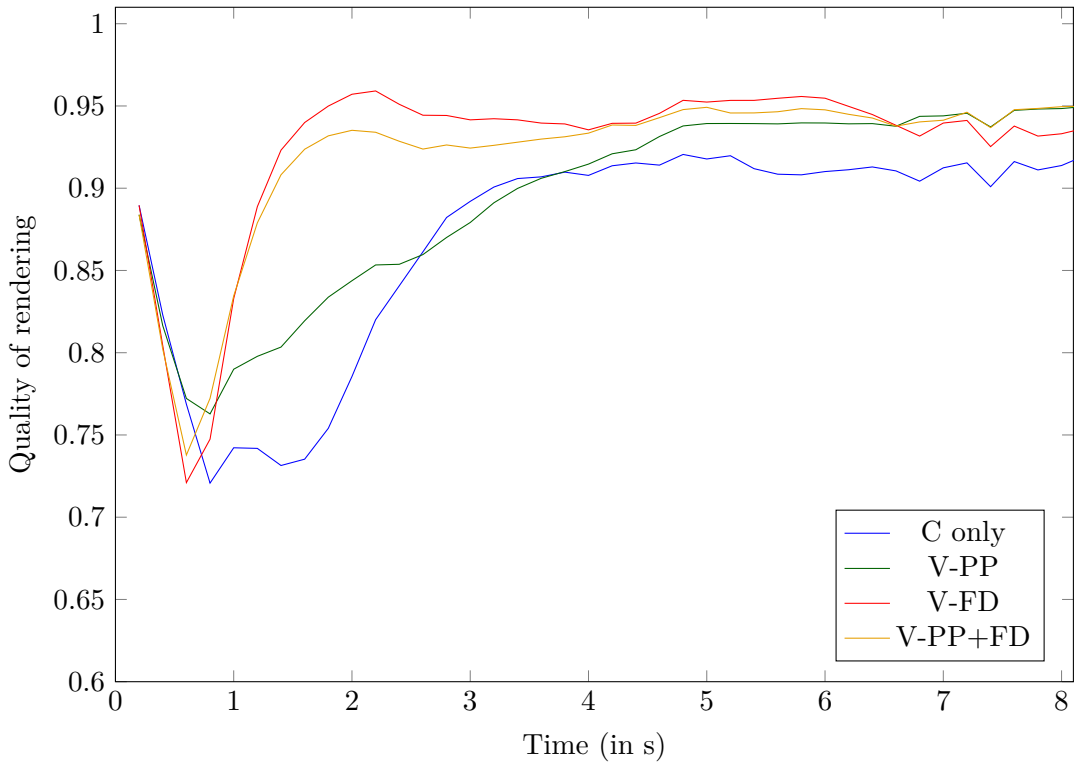


Figure 3.6: Average percentage of the image pixels that are correctly rendered against time, for all users with bookmarks, and using a bandwidth (BW) of 1 Mbps. The origin, $t = 0$, is the time of the first click on a bookmark. Each curve corresponds to a streaming policy.

Figure 3.6 compares the quality of the view of a user after their first click on a bookmark. The ratio of pixels correctly displayed is computed in the client algorithm, see also

algorithm 3. In this figure we use a bandwidth of 1 Mbps. The blue curve corresponds to the culling policy. Clicking on a bookmark generates a user path with less spatial locality, causing a large drop in visual quality that is only compensated after 4 seconds. During the first second, the camera moves from the current viewport to the bookmarked viewport.

When the data has been prefetched according to the probability of the bookmark to be clicked, the drop in quality is less visible (V-PP curve). However, by benefiting from the precomputation of visible triangles and ordering of the important triangles in a bookmark (V-FD) the drop in quality is still there, but is very short (approximately four times shorter than for culling). This drop in quality is happening during the transition on the path. More quantitatively, with a 1 Mbps bandwidth, 3 seconds are necessary after the click to recover 90% of correct pixels.

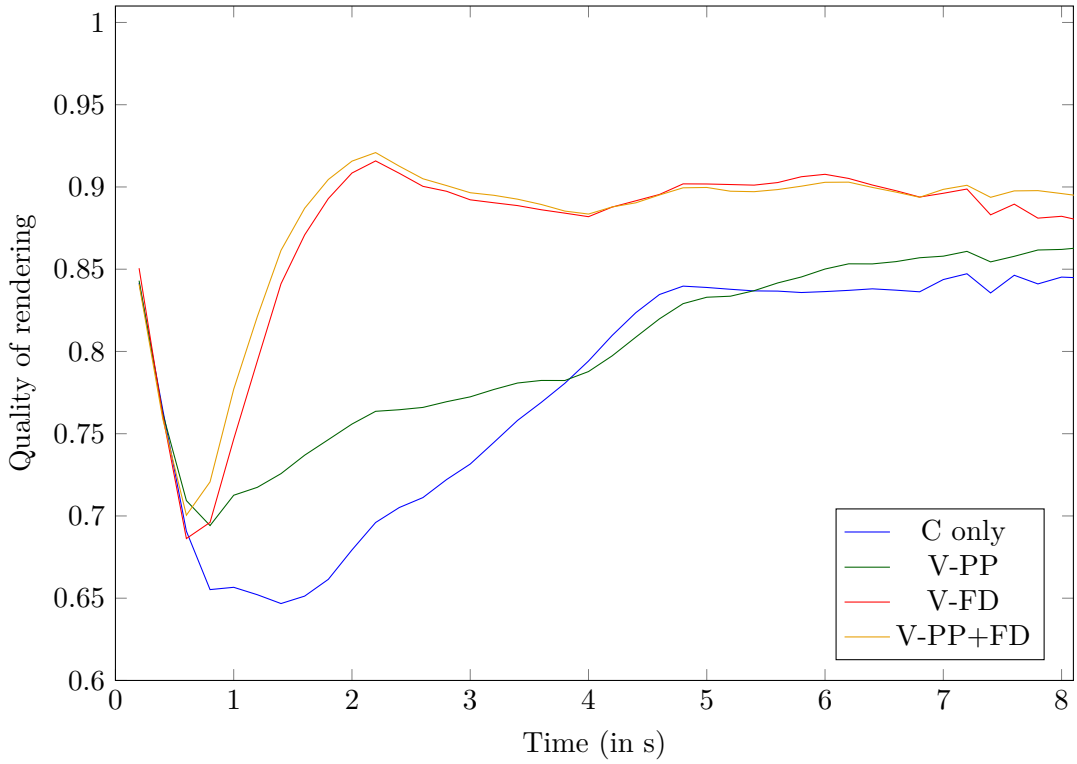


Figure 3.7: Average percentage of the image pixels that are correctly rendered against time –for all users with bookmarks, and using a bandwidth (BW) of 0.5 Mbps. The origin, $t = 0$, is the time of the first click on a bookmark. Each curve corresponds to a streaming policy.

Figure 3.7 showed the results of the same experiment with 0.5 Mbps bandwidth. Here, it takes 4 to 5 seconds to recover 85% of the pixels with culling and V-PP, against 1.5 second for recovering 90% with V-FD. Combining both strategies (V-PP+FD) leads to the best quality.

At 1 Mbps bandwidth, V-PP penalizes the quality, as the curve V-PP-FD leads to a

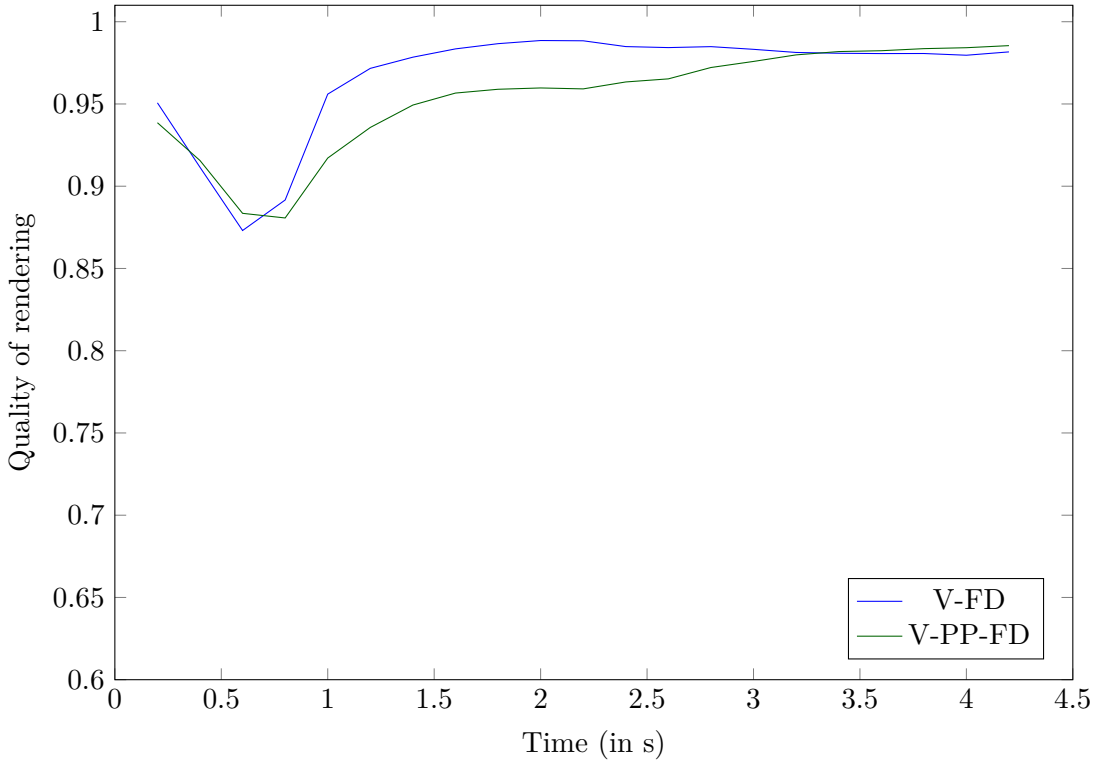


Figure 3.8: Same curve as Figures 3.6 and 3.7, for comparing streaming policies V-FD alone and V-PP+FD. BW=2Mbps

lower quality image than V-FD alone. This effect is even stronger when the bandwidth is set to 2 Mbps (Figure 3.8). Both streaming strategies based on the precomputation of the ordering improves the image quality. We see here, that V-FD has a greater impact than V-PP. Here, V-PP may prefetch content that eventually may not be used, whereas V-FD only sends relevant 3D content (knowing which bookmark has been just clicked).

We present only the results after the first click. For subsequent clicks, we found that other factors came into play and thus, it is hard to analyze the impact of the various streaming policies. For instance, a user may revisit a previously visited bookmark, or the bookmarks may overlap. If the users click on a subsequent bookmark after a long period, then more content would have been fetched for this user, making comparisons difficult.

To summarize, we found that exploiting the fact that bookmarked viewpoints are frequently visited to precompute the visible faces and sort them according to projected areas can lead to significant improvement in image quality after a user interaction (clicking on a bookmark). This alone can lead to 60% less triangles being sent, with 1/3 of the triangles sufficient to ensure 90% of pixels correctly rendered, compared to doing frustum/backface culling. If we fetch these precomputed faces of the destination viewpoint this way immediately after the click, during the “fly-to” transition, then we can already significantly improve the quality without any prefetching. Prefetching helps if the bandwidth is low,

and fewer triangles can be downloaded during this transition. The network conditions play a minimum role in this key message — bookmarking allows precomputation of an ordered list of visible faces, and this holds regardless of the underlying network condition (except for non-interesting extreme cases, such as negligible bandwidth or abundance of bandwidth).

3.4 Conclusion

In this chapter, we have described an interface that allows a user to navigate in a scene that is being streamed. We identified and addressed the problems linked to the dynamics of both the user behaviour and the 3D content.

- Navigating in a 3D scene can be complex, due to the many degrees of freedom, and tweaking the interface can increase the user's quality of experience.
- Adding bookmarks to the interface increases the quality of experience of the users and makes them visiting more data in the same amount of time.
- This increase in speed of navigation has a negative impact on the quality of service of the system.
- Having bookmarks in the scene biases the users navigation and makes the navigation more predictable: it is possible to link data utility to bookmarks in order to benefit from this predictability.

However, the system described in this chapter has some drawbacks and fails to answer the problems we mentioned in Section 1, and all these problems come from the fact that **the content preparation is inexistent**. The server knows all the data and simply determines what the client needs, it prepares the content and builds chunks on the go. Thus, the server has to keep track of what the client already has (which will eat memory) and has to compute what should be sent next (which will eat CPU). The scalability of such a server is therefore inexistent. Furthermore, we only considered geometry streaming: materials and textures are downloaded before the streaming starts, which causes great latency at the start of the streaming and harms the quality of experience.

After learning these lessons, we describe, in the next chapter, what can be done in order to alleviate these issues. We show how the standard for video steaming, DASH, teaches us to prepare 3D content in order to remove all server side computations, to elaborate great streaming policies, and to support both geometry and texture chunks.

Chapter 4

DASH-3D

Contents

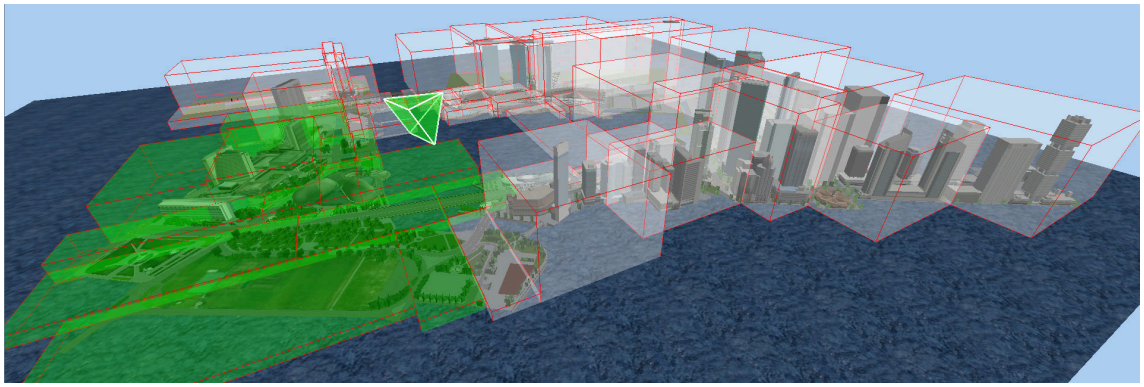


Figure 4.1: A subdivided 3D scene with a viewport and regions delimited with red edges. In white, the regions that are outside the field of view of the camera; in green, the regions inside the field of view of the camera.

Dynamic Adaptive Streaming over HTTP (DASH) is now a widely deployed standard for video streaming, and even though video streaming and 3D streaming are different problems, many of DASH features can inspire us for 3D streaming. In this chapter, we present the most important contribution of this thesis: adapting DASH to 3D streaming.

First, we show how to prepare 3D data into a format that complies with DASH data organization, and we store enough metadata to enable a client to perform efficient streaming. The data preparation consists in partitioning the scene into spatially coherent cells and segmenting each cell into chunks with a fixed number of faces, which are sorted by area so that faces of a different level of detail are not grouped together. We also export each texture at different resolutions. We encode the metadata that describes the data organization into a 3D version of the Media Presentation Description (MPD) that DASH uses for video. All this prepared content is then stored on a simple static HTTP server: a clients can request the content without any need for computation on the server side, allowing a server to support an arbitrary number of clients.

We then propose DASH-3D clients that are viewpoint aware: they perform frustum culling to eliminate cells outside the viewing volume of the camera (as shown in Figure 4.1). We define utility metrics to give a score to each chunk of data, be it geometry or texture, based on offline information that is given in the MPD, and online information that the client is able to compute, such as view parameters, user interaction or bandwidth measurements. We also define streaming policies that rely on those utilities in order for the client to determine which chunks need to be downloaded. We finally evaluate these system parameters under different bandwidth setups and compare our streaming policies.

4.1 Introduction

In this chapter, we take a little step back from interaction and propose a system with simple interactions that however, addresses most of the open problems mentioned in Section 1. We take inspiration from video streaming: working on the similarities between video streaming and 3D streaming (seen in 1.2), we benefit from the DASH efficiency (seen in 2.1.1) for streaming 3D content. DASH is based on content preparation and structuring which helps not only the streaming policies but also leads to a scalable and efficient system since it moves completely the load from the server to the clients. A DASH client downloads the structure of the content, and then, depending on its needs and independently of the server, decides what to download.

In this chapter, we show how to mimic DASH video with 3D streaming, and we develop a system that keeps DASH benefits. Section 4.2 describes our content preparation and metadata, and all the preprocessing that is done to our model to allow efficient streaming. Section 4.3 gives possible implementations of clients that exploit the content structure. Section 4.4 evaluates the impact of the different parameters that appear both in the content preparation and the client. Finally, Section 4.5 sums up our work and explains how it tackles the challenges raised in the conclusion of the previous chapter.

4.2 Content preparation

In this section, we describe how we preprocess and store the 3D data of the NVE, consisting of a polygon soup, textures, and material information into a DASH-compliant Media Presentation Description (MPD) file. In our work, we use the `obj` file format for the polygons, `png` for textures, and `mtl` format for material information. The process, however, applies to other formats as well.

4.2.1 The MPD File

In DASH, the information about content storage and characteristics, such as location, resolution, or size, is extracted from an MPD file by the client. The client relies only on this information to decide which chunk to request and at which quality level. The MPD file is an XML file that is organized into different sections hierarchically. The period element is a top-level element, which for the case of video, indicates the start time and length of a video chapter. This element does not apply to NVE, and we use a single period for the whole scene, as the scene is static. Each period element contains one or more adaptation sets, which describe the alternate versions, formats, and types of media. We utilize adaptation sets to organize a 3D scene's material, geometry, and texture.

The piece of software that does the preprocessing of the model consists in file manipulation and is written in Rust. It successively preprocesses the geometry and then the textures. The MPD is generated by a library named xml-rs[1] which works like a stack:

- a structure is created on the root of the MPD file;
- the `start_element` method creates a new child in the XML file;
- the `end_element` method ends the current child and pops the stack.

This structure is passed along with our geometry and texture preprocessors that can add elements to the XML file as they are generating the corresponding data chunks.

4.2.2 Adaptation sets

When the user navigates freely within an NVE, the frustum at given time almost always contains a limited part of the 3D scene. Similar to how DASH for video streaming partitions a video clip into temporal chunks, we segment the polygons into spatial chunks, such that the DASH client can request only the relevant chunks.

Geometry management

We use a space partitioning tree to organize the faces into cells. A face belongs to a cell if its barycenter falls inside the corresponding bounding box. Each cell corresponds to an adaptation set. Thus, geometry information is spread on adaptation sets based on spatial coherence, allowing the client to download the relevant faces selectively. A cell is relevant if it intersects the frustum of the client's current viewpoint. Figure 4.1 shows the relevant cells in green. As our 3D content, a virtual environment, is biased to spread along the horizontal plane, we split the bounding box alternatively along the two horizontal directions.

We create a separate adaptation set for large faces (e.g., the sky or ground) because they are essential to the 3D model and do not fit into cells. We consider a face to be large if its area in 3D is more than $a + 3\sigma$, where a and σ are the average and the standard deviation of 3D area of faces respectively. In our example, it selects the 5 largest faces that represent 15% of the total face area. We thus obtain a decomposition of the NVE into adaptation sets that partitions the geometry of the scene into an adaptation that contains the larger faces of the model, and smaller adaptation sets containing the remaining faces.

We store the spatial location of each adaptation set, characterized by the coordinates of its bounding box, in the MPD file as the supplementary property of the adaptation set in the form of "$x_{\min}$, *width*, $y_{\min}$, *height*, $z_{\min}$, *depth*" (as shown in Snippet 4.1). This information is used by the client to implement a view-dependent streaming (Section 4.3).

[1] https://github.com/netvl/xml-rs

Texture management

As with geometry data, we handle textures using adaptation sets but separate from geometry. Each texture file is contained in a different adaptation set, with multiple representations providing different image resolutions (see Section 4.2.3). We add an attribute to each adaptation set that contains texture, describing the average color of the texture. The client can use this attribute to render a face for which the corresponding texture has not been loaded yet, so that most objects appear, at least, with a uniform natural color (see Figure 4.2).

Material management

The material (MTL) file is a text file that describes all materials used in the OBJ files for the entire 3D model. A material has a name, properties such as specular parameters, and, most importantly, a path to a texture file. The MTL file maps each face of the OBJ to a material. As the MTL file is a different type of media than geometry and texture, we define a particular adaptation set for this file, with a single representation.

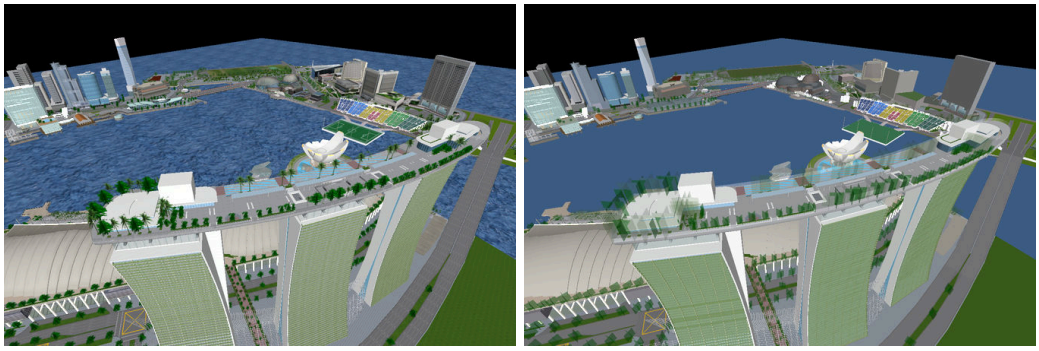
4.2.3 Representations

Each adaptation set can contain one or more representations of the geometry or texture data, at different levels of detail (e.g., a different number of faces). For geometry, the resolution (i.e., 3D areas of faces) is heterogeneous, thus applying a sensible multi-resolution representation is cumbersome: the 3D area of faces varies from 0.01 to more than $10K$, disregarding the outliers. For textured scenes, it is common to have such heterogeneous geometry size since information can be stored either in geometry or texture. Thus, handling the streaming compromise between geometry and texture is more adaptive than handling separately multi-resolution geometry. Moreover, as our faces are partitioned into independent cells, multi-resolution would cause difficult stitching issues such as topological gaps between the cells.

For an adaptation set containing texture, each representation contains a single segment where the image file is stored at the chosen resolution. In our example, from the full-size image, we generate successive resolutions by dividing both height and width by 2, stopping when the image size is less or equal to 64×64. Figure 4.2 illustrates the use of the textures against the rendering using a single, average color per face.

4.2.4 Segments

To allow random access to the content within an adaptation set storing geometry data, we group the faces into segments. Each segment is then stored as an OBJ file which can be individually requested by the client. For geometry, we partition the faces in an adaptation



(a) With full resolution textures (b) With average colors

Figure 4.2: Rendering of the model with different styles of textures

set into sets of N_s faces, by first sorting the faces by their area in 3D space in descending order, and then place each successive N_s faces into a segment. Thus, the first segment contains the biggest faces and the last one the smallest. In addition to the selected faces, a segment stores all face vertices and attributes so that each segment is independent. For textures, each representation contains a single segment.

Now that the 3D data is partitioned and that the MPD file is generated, we see in the next section how the client uses the MPD to request the appropriate data chunks.

4.3 Client

In this section, we specify a DASH NVE client which exploits the preparation of the 3D content in an NVE for streaming.

The generated MPD file describes the content organization so that the client gets all the necessary information to make educated decisions and query the 3D content it needs according to the available resources and current viewpoint. A camera path generated by a particular user is a set of viewpoint $v(t_i)$ indexed by a continuous time interval $t_i \in [t_1, t_{end}]$.

All DASH clients are built from the same basic bricks, as shown in Figure 4.3:

- the *access client*, which is the module that deals with making HTTP requests and receiving responses;
- the *segment parsers*, which decode the data downloaded by the access client, whether it be materials, geometry or textures;
- the *control engine*, which analyses the bandwidth to dynamically adapt to it;
- the *media engine*, which renders the multimedia content and the user interface to the screen.

The DASH client first downloads the MPD file to get the material file containing infor-

```
<AdaptationSet>
    <SupplementalProperty value="-8834.11230,2201.58853,
        -0.16950, 174.81540,-1344.47740,4767.83367" />
 <BaseURL>as1/</BaseURL>
 <Representation>
  <BaseURL>repr1/</BaseURL>
  <SegmentList>
   <SegmentURL area="2540342.3" size="120K" media="s0.obj" />
   <SegmentURL area="1124.4" size="162K" media="s1.obj" />
   <SegmentURL area="412.6" size="173K" media="s2.obj" />
   <SegmentURL area="270.3" size="147K" media="s3.obj" />
  </SegmentList>
 </Representation>
</AdaptationSet>

<AdaptationSet area="198632.73912" average="178,176,173" mimeType="image/png">
    <BaseURL>textures/MFLOOR07.PNG/</BaseURL>
    <Representation>
        <BaseURL>64x64/</BaseURL>
        <SegmentList>
            <SegmentURL size="7K" mse="57.6" media="t.png" />
        </SegmentList>
    </Representation>
    <Representation>
        <BaseURL>128x128/</BaseURL>
        <SegmentList>
            <SegmentURL size="27K" mse="0.0" media="t.png" />
        </SegmentList>
    </Representation>
</AdaptationSet>
```

Snippet 4.1: MPD description of a geometry adaptation set, and a texture adaptation set.

mation about all the geometry and textures available for the entire 3D model. At time instance t_i, the DASH client decides to download the appropriate segments containing the geometry and the texture to generate the viewpoint $v(t_{i+1})$ for the time instance t_{i+1}.

Starting from t_1, the camera continuously follows a camera path $C = \{v(t_i), t_i \in [t_1, t_{end}]\}$, along which downloading opportunities are strategically exploited to sequentially query the most useful segments.

4.3.1 Segment utility

Unlike video streaming, where the bitrate of each segment correlates with the quality of the video received, for 3D content, the size (in bytes) of the content does not necessarily correlate well to its contribution to visual quality. A large polygon with huge visual impact takes the same number of bytes as a tiny polygon. Further, the visual impact is *view dependent* — a large object that is far away or out of view does not contribute to

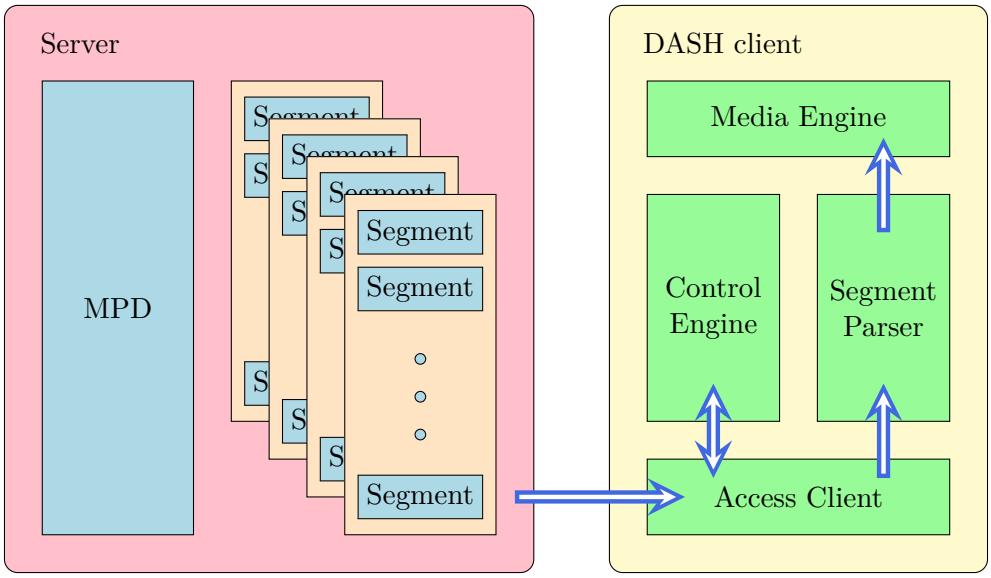


Figure 4.3: DASH client-server architecture

the visual quality as much as a smaller object that is closer to the user. As such, it is important for a DASH-based NVE client to estimate the usefulness of a given segment to download, so that it can make good decisions about what to download. We call this usefulness the *utility* of the segment.

The utility is a function of a segment, either geometry or texture, and the current viewpoint (camera location, view angle, and look-at point), and is therefore dynamically computed online by the client from parameters in the MPD file.

Offline parameters

Let us detail first, all parameters available from the offline/static preparation of the 3D NVE. These parameters are stored in the MPD file. First, for each geometry segment s^G there is a predetermined 3D area $\mathcal{A}\left(s^G\right)$, equal to the sum of all triangle areas in this segment (in 3D); it is computed as the segments are created. Note that the texture segments have similar information, but computed at *navigation time* t_i. The second information stored in the MPD for all segments, geometry, and texture, is the size of the segment (in kB).

Finally, for each texture segment s^T, the MPD stores the *MSE* (mean square error) of the image and resolution, relative to the highest resolution (by default, triangles are filled with its average color). Offline parameters are stored in the MPD as shown in Snippet 4.1.

Online parameters

In addition to the offline parameters stored in the MPD file for each segment, view-dependent parameters are computed at navigation time. First, a measure of 3D area is

computed for texture segments. As a texture maps on a set of triangles, we account for the area in 3D of all these triangles. We could consider such an offline measure (attached to the adaptation set containing the texture), but we prefer to only account for the triangles that have been already downloaded by the client. We call the set of triangles colored by a texture T: $\Delta(s^T) = \Delta(T)$ (depending only on T and equal for any representation/segment s^T in this texture adaptation set). At each time t_i, a subset of $\Delta(T)$ has been downloaded; we denote it $\Delta(T, t_i)$.

Moreover, each geometry segment belongs to a geometry adaptation set AS^G whose bounding box coordinates are stored in the MPD. Given the coordinates of the bounding box $\mathcal{BB}(AS^G)$ and the viewpoint $v(t_i)$ at time t_i, the client computes the distance $\mathcal{D}(v(t_i), AS^G)$ of the bounding box $\mathcal{BB}(AS^G)$ as the distance from the center of $\mathcal{BB}(AS^G)$ to the principal point of the camera, given in $v(t_i)$.

Utility for geometry segments

We now have all parameters to derive a utility measure of a geometry segment. Utility for texture segments follows from the geometric utility.

The utility of a geometric segment s^G for a viewpoint $v(t_i)$ is:

$$\mathcal{U}\Big(s^G, v(t_i)\Big) = \frac{\mathcal{A}(s^G)}{\mathcal{D}(v(t_i), AS^G)^2}$$

where AS^G is the adaptation set containing s^G.

Basically, the utility of a segment is proportional to the area that its faces cover, and inversely proportional to the square of the distance between the camera and the center of the bounding box of the adaptation set containing the segment. That way, we favor segments with big faces that are close to the camera.

Utility for texture segments

For a texture T stored in a segment s^T, the triangles in $\Delta(T)$ are stored in arbitrary geometry segments, that is, they do not have spatial coherence. Thus, for each k^{th} downloaded geometry segment s^G_k, and total downloaded segment K at time t_i, we collect the triangles of $\Delta(T, t_i)$ in s^G_k, and compute the ratio of $\mathcal{A}(s^G_k)$ covered by these triangles. So, we define the utility:

$$\mathcal{U}\Big(s^T, v(t_i)\Big) = psnr(s^T) \sum_{k \in K} \frac{\mathcal{A}(s^G_k \cap \Delta(T, t_i))}{\mathcal{A}(s^G_k)} \mathcal{U}\Big(s^G_k, v(t_i)\Big)$$

where we sum over all geometry segments received before time t_i that intersect $\Delta(T, t_i)$ and such that the adaptation set it belongs to is in the frustum. This formula defines the utility of a texture segment by computing the linear combination of the utility of the geometry segments that use this texture, weighted by the proportion of area covered by the

texture in the segment. We compute the PSNR by using the MSE in the MPD and denote it $psnr(s^T)$. We do this to acknowledge the fact that a texture at a greater resolution has a higher utility than a lower resolution texture. The equivalent term for geometry is 1 (and does not appear). Having defined a utility on both geometry and texture segments, the client uses it next for its streaming strategy.

4.3.2 DASH adaptation logic

Along the camera path $C = \{v(t_i)\}$, viewpoints are indexed by a continuous time interval $t_i \in [t_1, t_{end}]$. Contrastingly, the DASH adaptation logic proceeds sequentially along a discrete time line. The first HTTP request made by the DASH client at time t_1 selects the most useful segment s_1^* to download and will be followed by subsequent decisions at $t_2, t_3, \ldots$. While selecting s_i^*, the i^{th} best segment to request, the adaptation logic compromises between geometry, texture, and the available representations given the current bandwidth, camera dynamics, and the previously described utility scores. The difference between t_{i+1} and t_i is the s_i^* delivery delay. It varies with the segment size and network conditions. Algorithm 5 details how our DASH client makes decisions.

Algorithm 5: Algorithm to identify the next segment to query

input : Current index i, time t_i, viewpoint $v(t_i)$, buffer of already downloaded `segments` $\mathcal{B}_i$, MPD, utility metric $\mathcal{U}$, streaming policy Ω
output: Next segment s_i^* to request, updated buffer $\mathcal{B}_{i+1}$

```
(bw_estimation, rtt_estimation) ← estimate_network_parameters();
candidates ← all_segments
        .filter(segment → segment ∉ downloaded_segments)
        .filter(segment → segment ∈ frustum);
best_segment ← argmax(candidates, segment → Ω(U, segment));
downloaded_segments.append(best_segment);
```

A naive way to sequentially optimize the utility $\mathcal{U}$ is to limit the decision-making to the current viewpoint $v(t_i)$. In that case, the best segment s to request would be the one maximizing $\mathcal{U}(s, v(t_i))$ to simply make a better rendering from the current viewpoint $v(t_i)$. Due to transmission delay however, this segment will be only delivered at time $t_{i+1} = t_{i+1}(s)$ depending on the segment size and network conditions:

$$t_{i+1}(s) = t_i + \frac{\texttt{size}(s)}{\widehat{BW_i}} + \widehat{\tau_i}$$

In consequence, the most useful segment from $v(t_i)$ at decision time t_i might be less useful at delivery time from $v(t_{i+1})$.

A better solution is to download a segment that is expected to be the most useful in the

future. With a temporal horizon χ, we can optimize the cumulated $\mathcal{U}$ over $[t_{i+1}(s), t_i + \chi]$:

$$s_i^* = \underset{s \in \mathcal{S} \setminus \mathcal{B}_i \cap \mathcal{FC}}{\operatorname{argmax}} \int_{t_{i+1}(s)}^{t_i+\chi} \mathcal{U}(s, \hat{v}(t_i))dt \tag{4.1}$$

In our experiments, we typically use $\chi = 2s$ and estimate the (4.1) integral by a Riemann sum where the $[t_{i+1}(s), t_i + \chi]$ interval is divided in 4 subintervals of equal size. For each subinterval extremity, an order 1 predictor $\hat{v}(t_i)$ linearly estimates the viewpoint based on $v(t_i)$ and speed estimation (discrete derivative at t_i).

We also tested an alternative greedy heuristic selecting the segment that optimizes an utility variation during downloading (between t_i and t_{i+1}):

$$s_i^{\texttt{GREEDY}} = \underset{s \in \mathcal{S} \setminus \mathcal{B}_i \cap \mathcal{FC}}{\operatorname{argmax}} \frac{\mathcal{U}\Big(s, \hat{v}(t_{i+1}(s))\Big)}{t_{i+1}(s) - t_i} \tag{4.2}$$

4.3.3 JavaScript client

In order to be able to evaluate our system, we need to collect traces and perform analyses on them. Since our scene is large, and since the system we are describing allows navigating in a streaming scene, we developed a JavaScript web client that implements our utility metrics and policies.

Media engine

Performance of our system is a key aspect in our work; as such, we can not use the default geometries described in Section 1.3.1 because of their poor performance, and we instead use buffer geometries. However, in our system, the way changes happen to the 3D content is always the same: we only add faces and textures to the model. We therefore implemented a class that derives `BufferGeometry`, for more convenience.

- It has a constructor that takes as parameter the number of faces: it allocates all the memory needed for our buffers so we do not have to reallocate it later (which would be inefficient).
- It keeps track of the number of faces it is currently holding: it can then avoid rendering faces that have not been filled and knows where to add new faces.
- It provides a method to add a new polygon to the geometry.
- It also keeps track of what part of the buffers has been transmitted to the GPU: THREE.js allows us to set the range of the buffer that we want to update, and we are able to update only what is necessary.

4.3.4 Our 3D model class.

As said in the previous subsections, a geometry and a material are bound together in a mesh. This means that we are forced to have as many meshes as there are materials in our model. To make this easy to manage, we implemented a **Model** class, that holds both geometry and textures. We can add vertices, faces, and materials to this model, and it internally manages the right geometries, materials and meshes. In order to avoid having many models that share the same material (which would harm performance), it automatically merges faces that share the same material in the same buffer geometry, as shown in Figure 4.4.

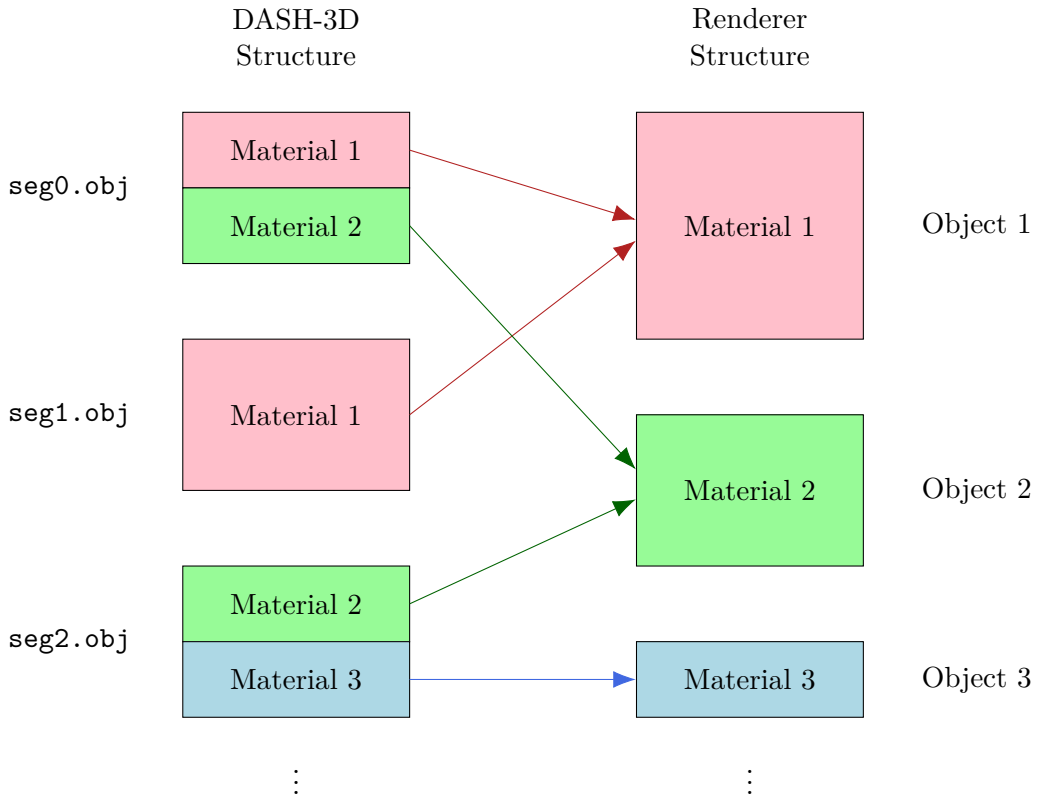


Figure 4.4: Reordering of the content on the renderer

Access client

In order to be able to implement our view-dependent DASH-3D client, we need to implement the access client, which is responsible for deciding what to download and for downloading it. To do so, we use the strategy pattern illustrated in Figure 4.5. We maintain a base class named `LoadingPolicy` that contain some attributes and functions to keep track of what has been downloaded. This class exposes a function named `nextSegment` that takes two arguments:

- the MPD, so that a strategy can know all the metadata of the segments before making its decision;
- the camera, because the next best segment depends on the camera position.

The greedy and proposed policies from the previous chapter are all classes that derive from `LoadingPolicy`. Then, the main class responsible for the loading of segments is the `DashLoader` class. It uses `XMLHttpRequest`s, which are the usual way of making HTTP requests in JavaScript, and it calls the corresponding parser on the results of those requests. The `DashLoader` class accepts as parameter a function that will be called each time some data has been downloaded and parsed: this data can contain vertices, texture coordinates, normals, materials or textures, and they can all be added to the `Model` class that we described in Section 4.3.4.

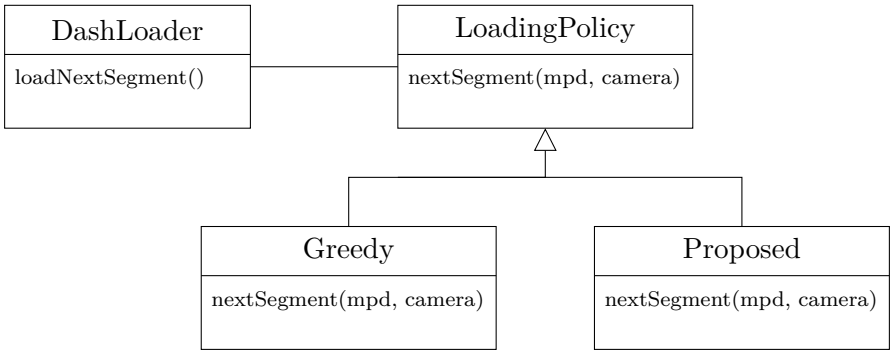


Figure 4.5: Class diagram of our DASH client

Performance

JavaScript requires the use of *web workers* to perform parallel computing. A web worker is a script in JavaScript that runs in the background, on a separate thread and that can communicate with the main script by sending and receiving messages. Since our system has many tasks to perform, it is natural to use workers to manage the streaming without impacting the framerate of the renderer. However, what a worker can do is very limited, since it cannot access the variables of the main script. Because of this, we are forced to run the renderer on the main script, where it can access the HTML page, and we move all the other tasks (i.e. the access client, the control engine and the segment parsers) to the worker. Since the main script is the only thread communicating with the GPU, it will still have to update the model with the parsed content it receives from the worker. We do not use web workers to improve the framerate of the system, but rather to reduce the latency that occurs when receiving a new segment, which can be frustrating in a single thread scenario, since each time a segment is received, the interface would freeze for around half a second. A sequence diagram of what happens when downloading, parsing and rendering content is shown in Figure 4.6.

4.3.5 Rust client

However, a web client is not sufficient to analyse our streaming policies: many tasks are performed (such as rendering, and managing the interaction) and all this overhead pollutes the analysis of our policies. This is why we also implemented a client in Rust, for simulation, so we can gather precise simulated data.

Our requirements are quite different that the ones we had to deal with in our JavaScript implementation. In this setup, we want to build a system that is the closest to our theoretical concepts. Therefore, we do not have a full client in Rust (meaning an application to which you would give the URL to an MPD file and that would allow you to navigate in the scene while it is being downloaded). In order to be able to run simulations, we develop the bricks of the DASH client separately: the access client and the media engine are totally isolated:

- the **simulator** takes a user trace as a parameter, it then replays the trace using specific parameters of the access client and outputs a file containing the history of the simulation (which files have been downloaded, and when);
- the **renderer** takes the user trace as well as the history generated by the simulator as parameters, and renders images that correspond to what would have been seen.

When simulating experiments, we run the simulator on many traces that we collected during user-studies, and we then run the renderer program according to the traces to generate images corresponding to the simulation. We are then able to compute PSNR between those frames and the ground truth frames. Doing so guarantees us that our simulator is not affected by the performances of our renderer.

4.4 Evaluation

We now describe our setup and the data we use in our experiments. We present an evaluation of our system and a comparison of the impact of the design choices we introduced in the previous sections.

4.4.1 Experimental setup

Model

We use a city model of the Marina Bay area in Singapore in our experiments. The model came in 3DS Max format and has been converted into Wavefront OBJ format before the processing described in Section 4.2. The converted model has 387,551 vertices and 552,118 faces. Table 4.1 gives some general information about the model and Figure 4.7 illustrates the heterogeneity of our model (wireframe rendering is used to illustrate the heterogeneity

of the geometry complexity). We partition the geometry into a k-d tree until the leafs have less than 10000 faces, which gives us 64 adaptation sets, plus one containing the large faces.

Files	Size
3DS Max	55 MB
OBJ file	62 MB
MTL file	0.27MB
Textures (high res)	167 MB
Textures (low res)	11 MB

Table 4.1: Sizes of the different files of the model

User navigations

To evaluate our system, we collected realistic user navigation traces which we can replay in our experiments. We presented six users with a web interface, on which the model was loaded progressively as the user could interact with it. The available interactions were inspired by traditional first-person interactions in video games, i.e., W, A, S, and D keys to translate the camera, and mouse to rotate the camera. We asked users to browse and explore the scene until they felt they had visited all important regions. We then asked them to produce camera navigation paths that would best present the 3D scene to a user that would discover it. To record a path, the users first place their camera to their preferred starting point, then click on a button to start recording. Every 100ms, the position, viewing angle of the camera and look-at point are saved into an array which will then be exported into JSON format. The recorded camera trace allows us to replay each camera path to perform our simulations and evaluate our system. We collected 13 camera paths this way.

Network setup

We tested our implementation under three network bandwidth of 2.5 Mbps, 5 Mbps, and 10 Mbps with an RTT of 38 ms, following the settings from DASH-IF [Forum, 2014]. The values are kept constant during the entire client session to analyze the difference in magnitude of performance by increasing the bandwidth.

In our experiments, we set up a virtual camera that moves along a navigation path, and our access engine downloads segments in real time according to Algorithm 5. We log in a JSON file the time when a segment is requested and when it is received. By doing so, we avoid wasting time and resources to evaluate our system while downloading segments and store all the information necessary to plot the figures introduced in the subsequent sections.

Hardware and software

The experiments were run on an Acer Aspire V3 with an Intel Core i7 3632QM processor and an NVIDIA GeForce GT 740M graphics card. The DASH client is written in Rust[2], using Glium[3] for rendering, and reqwest[4] to load the segments.

Metrics

To objectively evaluate the quality of the resulting rendering, we use PSNR. The scene as rendered offline using the same camera path with all the geometry and texture data available is used as ground truth. Note that a pixel error can occur in our case only in two situations: (i) when a face is missing, in which case the color of the background object is shown, and (ii) when a texture is either missing or downsampled. We do not have pixel error due to compression.

Experiments

We present experiments to validate our implementation choices at every step of our system. We replay the user-generated camera paths with various bandwidth conditions while varying key components of our system.

Table 4.2 sums up all the components we varied in our experiments. We compare the impact of two space-partitioning trees, a k-d tree and an octree, on content preparation. We also try several utility metrics for geometry segments: an offline one, which assigns to each geometry segment s^G the cumulated 3D area of its belonging faces $\mathcal{A}(s^G)$; an online one, which assigns to each geometry segment the inverse of its distance to the camera position; and finally our proposed method, as described in Section 4.3.1 $(\mathcal{A}(s^G)/\mathcal{D}(v(t_i), AS^G)^2)$. We consider two streaming policies to be applied by the client, proposed in Section 4.3. The greedy strategy determines, at each decision time, the segment that maximizes its predicted utility at arrival divided by its predicted delivery delay, which corresponds to equation (4.2). The second streaming policy that we run is the one we proposed in equation (4.1). We have also analyzed the effect of grouping the faces in geometry segments of an adaptation set based on their 3D area. Finally, we try several bandwidth parameters to study how our system can adapt to varying network conditions.

4.4.2 Experimental results

Figure 4.8 shows how the space partition can affect the rendering quality. We use our proposed utility metrics (see Section 4.3.1) and streaming policy from equation (4.1), on content divided into adaptation sets obtained either using a k-d tree or an octree

[2] https://www.rust-lang.org/
[3] https://github.com/glium/glium
[4] https://github.com/seanmonstar/reqwest/

Parameters	Values
Content preparation	Octree, k-d tree
Utility	Offline, Online, Proposed
Streaming policy	Greedy, Proposed
Grouping of Segments	Sorted based on area, Unsorted
Bandwidth	2.5 Mbps, 5 Mbps, 10 Mbps

Table 4.2: Different parameters in our experiments

and run experiments on all camera paths at 5 Mbps. The octree partitions content into non-homogeneous adaptation sets; as a result, some adaptation sets may contain smaller segments, which contain both important (large) and non-important polygons. For the k-d tree, we create cells containing the same number of faces N_a (here, we take $N_a = 10000$). Figure 4.8 shows that the system seems to be slightly less efficient with an octree than with a k-d tree based partition, but this result is not significant. For the remaining experiments, partitioning is based on a k-d tree.

Figure 4.9 displays how a utility metric should take advantage of both offline and online features. The experiments consider k-d tree cell for adaptation sets and the proposed streaming policy, on all camera paths. We observe that a purely offline utility metric leads to poor PSNR results. An online-only utility improves the results, as it takes the user viewing frustum into consideration, but still, the proposed utility (in Section 4.3.1) performs better.

Figure 4.10 shows the effect of grouping the segments in an adaptation set based on their area in 3D. The PSNR significantly improves when the 3D area of faces is considered for creating the segments. Since all segments are of the same size, sorting the faces by area before grouping them into segments leads to a skew distribution of how useful the segments are. This skewness means that the decision that the client makes (to download those with the largest utility first) can make a bigger difference in the quality.

We also compared the greedy vs. proposed streaming policy (as shown in Figure 4.11) for limited bandwidth (5 Mbps). The proposed scheme outperforms the greedy during the first 30s and does a better job overall. Table 4.3 shows the average PSNR for the proposed method and the greedy method for different downloading bandwidth. In the first 30 sec, since there are relatively few 3D contents downloaded, making a better decision at what to download matters more: we observe during that time that the proposed method leads to 1 — 1.9 dB better in quality terms of PSNR compared to the greedy method.

Table 4.4 shows the distribution of texture resolutions that are downloaded by greedy and our proposed scheme, at different bandwidths. Resolution 5 is the highest and 1 is the lowest. The table shows a weakness of the greedy policy: the distributioon of downloaded textures does not adapt to the bandwidth. In contrast, our proposed streaming policy adapts to an increasing bandwidth by downloading higher resolution textures (13.9% at

10 Mbps, vs. 0.3% at 2.5 Mbps). In fact, an interesting feature of our proposed streaming policy is that it adapts the geometry-texture compromise to the bandwidth. The textures represent 57.3% of the total amount of downloaded bytes at 2.5 Mbps, and 70.2% at 10 Mbps. In other words, our system tends to favor geometry segments when the bandwidth is low, and favor texture segments when the bandwidth increases.

	First 30 Sec			**Overall**		
BW (in Mbps)	2.5	5	10	2.5	5	10
Greedy	14.4	19.4	22.1	19.8	26.9	29.7
Proposed	16.3	20.4	23.2	23.8	28.2	31.1

Table 4.3: Average PSNR, Greedy vs. Proposed

Resolutions	**2.5 Mbps**	**5 Mbps**	**10 Mbps**
1	5.7% vs 1.4%	6.3% vs 1.4%	6.17% vs 1.4%
2	10.9% vs 8.6%	13.3% vs 7.8%	14.0% vs 8.3%
3	15.3% vs 28.6%	20.1% vs 24.3%	20.9% vs 22.5%
4	14.6% vs 18.4%	14.4% vs 25.2%	14.2% vs 24.1%
5	11.4% vs 0.3%	11.1% vs 5.9%	11.5% vs 13.9%

Table 4.4: Percentages of downloaded bytes for textures from each resolution, for the greedy streaming policy (left) and for our proposed scheme (right)

4.5 Conclusion

Our work in this chapter started with the question: can DASH be used for NVE? The answer is *yes*. In answering this question, we contributed by showing how to organize a polygon soup and its textures into a DASH-compliant format that (i) includes a minimal amount of metadata that is useful for the client, (ii) organizes the data to allow the client to get the most useful content first. We further show that the data organization and its description with metadata (precomputed offline) is sufficient to design and build a DASH client that is adaptive — it selectively downloads segments within its view, makes intelligent decisions about what to download, balances between geometry and texture while adapting to network bandwidth. This way, our system addresses the open problems we mentioned in Chapter 1.

- **It prepares and structures the content in a way that enables streaming**: all this preparation is precomputed, and all the content is structured according to DASH framework, geometry but also materials and textures. Furthermore, textures are

prepared in a multi-resolution manner, and even though multi-resolution geometry is not discussed here, the difficulty of integrating it in this system seem moderated: we could encode levels of detail in different representations and define a utility metric for each representation and the system should adapt naturally.

- **We are able to estimate the utility of each segment** by exploiting all the metadata given in the MPD and by analysing the camera parameters of the user.
- **We proposed a few streaming policies**, from the easiest to implement to the more complex, so that the client exploits the utility metrics to define a best guess for the next chunk to download.
- **The implementation is efficient**: the content preparation allows a client to get all the information it needs from metadata and the server has nothing else to do than to serve files. Special attention has been granted to the client's performance.

However, the work described in this chapter does not take any quality of experience metrics into account. We designed a 3D streaming system, but we kept the interaction system the simplest possible. Dealing with interaction while dealing with all of the other problems we try to solve seems hard, and we believe keeping the interaction simple was a necessary step to build a solid 3D streaming system. Now that we have this system, we are able to work again on the interaction problem and our work and conclusions are given in Chapter 5.

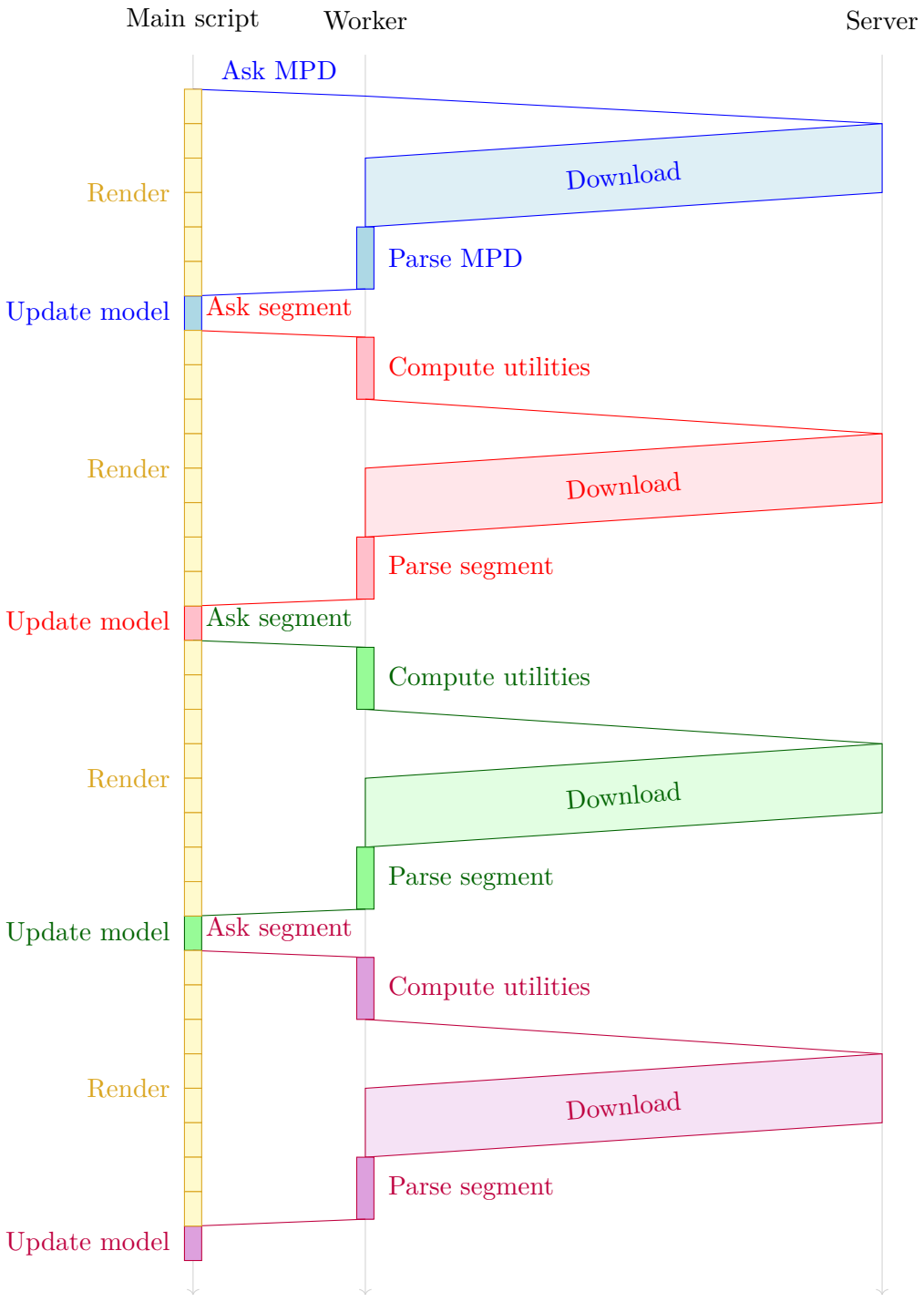
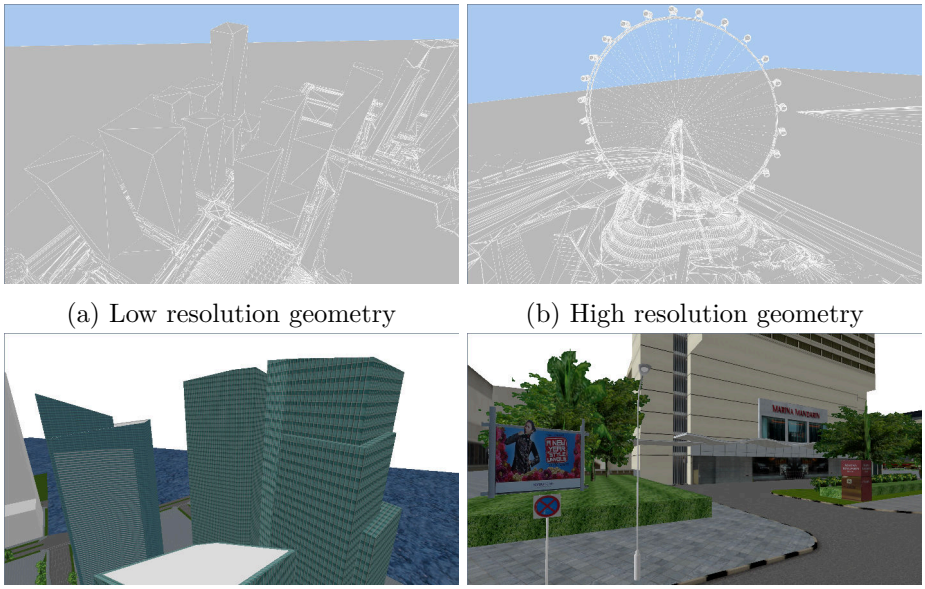


Figure 4.6: Repartition of the tasks on the main script and the worker



(a) Low resolution geometry (b) High resolution geometry

(c) Simplistic textures replicated (d) Detailed textures

Figure 4.7: Illustration of the heterogeneity of the model

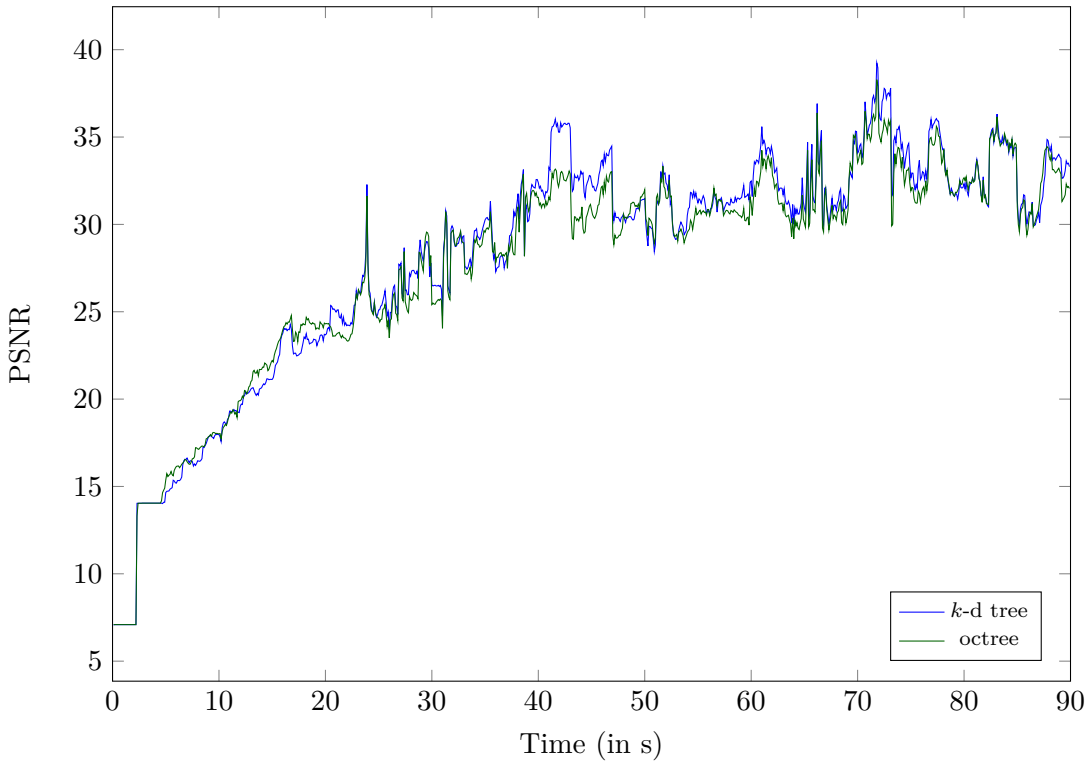


Figure 4.8: Impact of the space-partitioning tree on the rendering quality with a 5Mbps bandwidth.

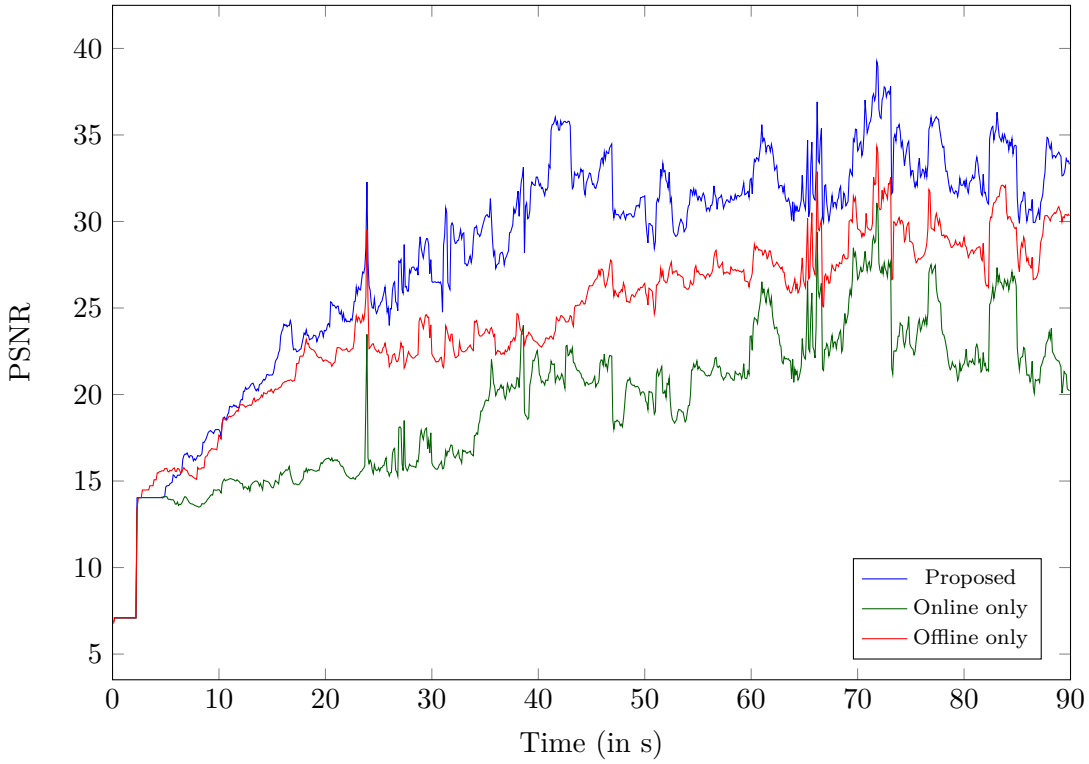


Figure 4.9: Impact of the segment utility metric on the rendering quality with a 5Mbps bandwidth.

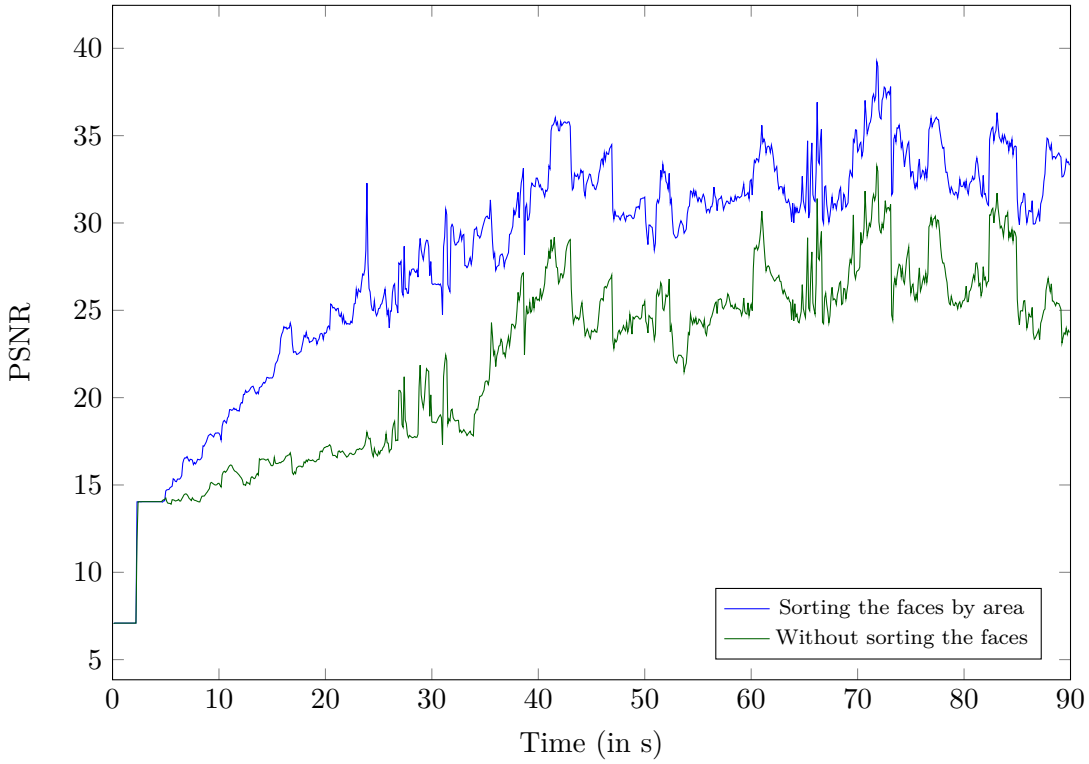


Figure 4.10: Impact of creating the segments of an adaptation set based on decreasing 3D area of faces with a 5Mbps bandwidth.

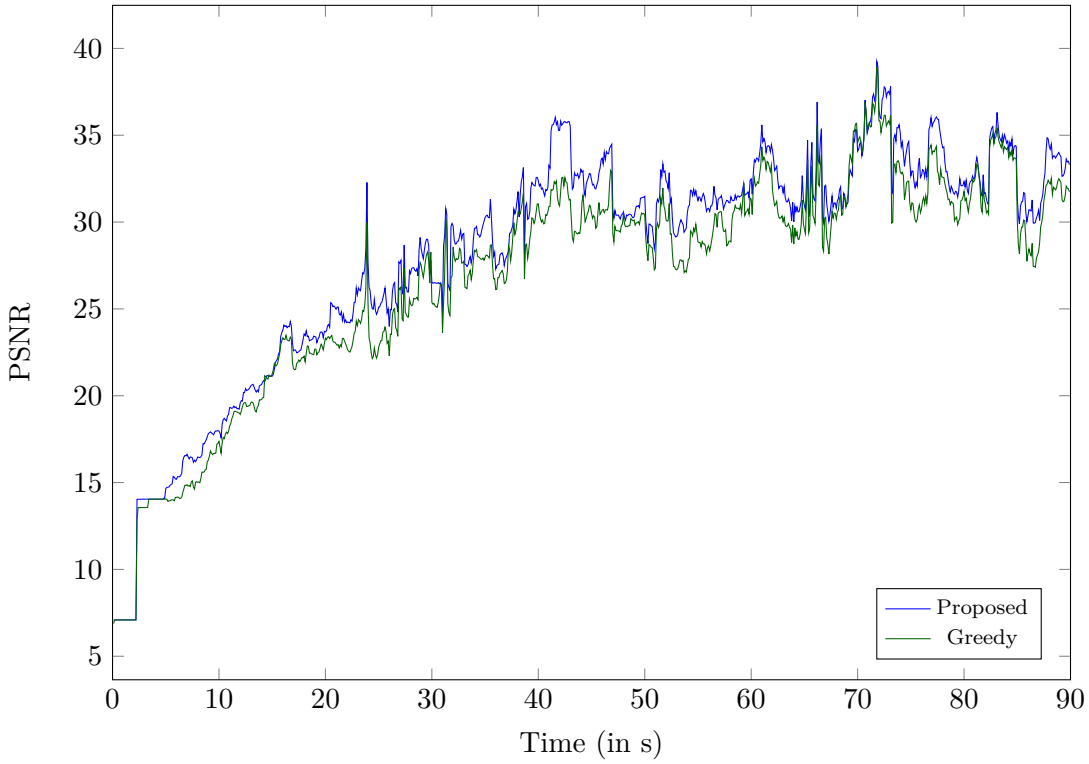


Figure 4.11: Impact of the streaming policy (greedy vs. proposed) with a 5 Mbps bandwidth.

Chapter 5

Bookmarks for DASH-3D on mobile devices

Contents

The growing capabilities and usage of mobile devices, especially smartphones, nowadays incur a progressive shift of many applications from desktop to mobile devices. In order to be made available and usable by the greater audience, 3D streaming and visualization should also be possible on mobile devices.

However, desktop devices tend to be much more powerful, have a larger memory and better network connections than mobile devices. In addition, the interactive modalities of these two types of devices are not comparable in any way: the desktop mostly uses keyboard and mouse, whereas most of the mobile devices only have a touchscreen, as well as various additional sensors (accelerometer, gyroscope, GPS, etc.). For these reasons, using DASH to stream 3D on mobile devices requires specific adaptations, that we describe in this chapter.

We add some widgets on the screen to support touch interactions: a virtual joystick is displayed on the screen and the user can touch it to translate the camera, instead of using the W, A, S and D keys on a computer keyboard. Since most mobile devices embed a gyroscope, we allow users to rotate the camera by physically rotating the device. This interaction is more precise and intuitive to the user, but it is also more tiring, this is why we also added a touch interaction to rotate the screen: a user can also "touch and drag" at any point on the screen that does not correspond to the joystick to rotate the camera. In order to ease navigation, we integrate bookmarks back, and we propose an enhanced version of the precomputations explained in Chapter 5 that we encode in the DASH Media Presentation Description. We then present a user study on 18 participants, which evaluates how users perceive the visual quality of the scene, and how their interactions affect it.

5.1 Introduction

In Chapter 3, we described how it is possible to modify a user interface to ease user navigation in a 3D scene, and how the system can benefit from it. In Chapter 4, we presented the DASH-3D streaming system, which does not depend on the interface nor on the user interaction. In this chapter, we will analyze how the user interaction can impact performances of DASH-3D. In order to do so, we follow these two steps based on our DASH framework:

- we design an interface allowing to navigate in a 3D scene on both desktop and mobile devices;
- we improve and adapt the bookmarks described in Chapter 3 to the context of DASH-3D and to mobile interaction.

In Section 5.2, we present the different choices we made for the interfaces, and we describe the new mobile interface. In Section 5.3, we describe how we embed the bookmarks into our DASH framework, and how we precompute data in order to improve the user quality of experience. In Section 5.4, we describe the user study we conducted, the data we collected and we analyse the results.

5.2 Desktop and mobile interactions

5.2.1 Desktop interaction

Regarding desktop interaction, we keep the interaction we described in Section 3.2.1, namely:

- W, A, S and D keys to translate the camera;
- mouse motions to rotate the camera.

A screenshot of this interface is displayed in Figure 5.1.

5.2.2 Mobile interaction

Mobile interactions are more complex because the user does not have the keyboard and mouse to interact with. However, there are some other sensors on most mobile devices that can help interaction. One useful sensor for 3D interaction on mobile devices is definitely the gyroscope. We use the gyroscope to enable a user to rotate his device to rotate the virtual camera. We also add the possibility to rotate the camera by using touch controls. The user can touch a part of the screen to get a hold at the virtual camera, and drag the camera direction along the two screen axis. This way, the user is not forced to perform a real-world half-turn to be able to look behind or to point the device towards the sky

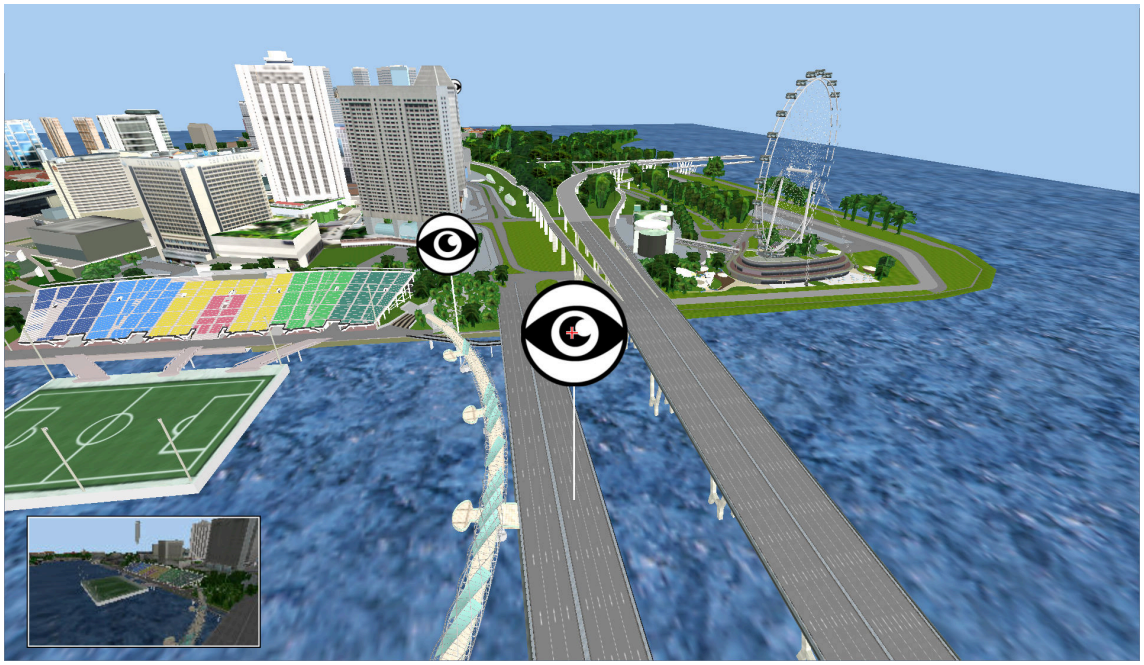


Figure 5.1: Screenshot of the desktop version, with a bookmark and its thumbnail on the bottom left corner and three bookmarks

(which can quickly become tiring) to look up. These interactions, however, only allow the user to rotate the camera but not translate it. For this reason, we display a small joystick on the bottom-left corner of the screen that mimics the first person video games interactions and allows the user travelling in the scene:

- moving the joystick up makes the camera move forward;
- moving the joystick down makes the camera move backwards;
- moving the joystick on the sides makes the camera move sideways.

A screenshot of this interface is displayed in Figure 5.2. The virtual joystick is rendered as a black circle inside a larger semi-transparent white circle. The black circle can be moved up, down, and sideways to define the direction in which the camera is translated.

5.3 Adding bookmarks into DASH NVE framework

While the previously defined interactions allow users to navigate freely throughout the scene, controlling such a high number of degrees of freedom can feel overwhelming to some users. That is why we introduce bookmarks, i.e. widgets that help the users reach a distant part of the scene using only a single, simple, interaction.

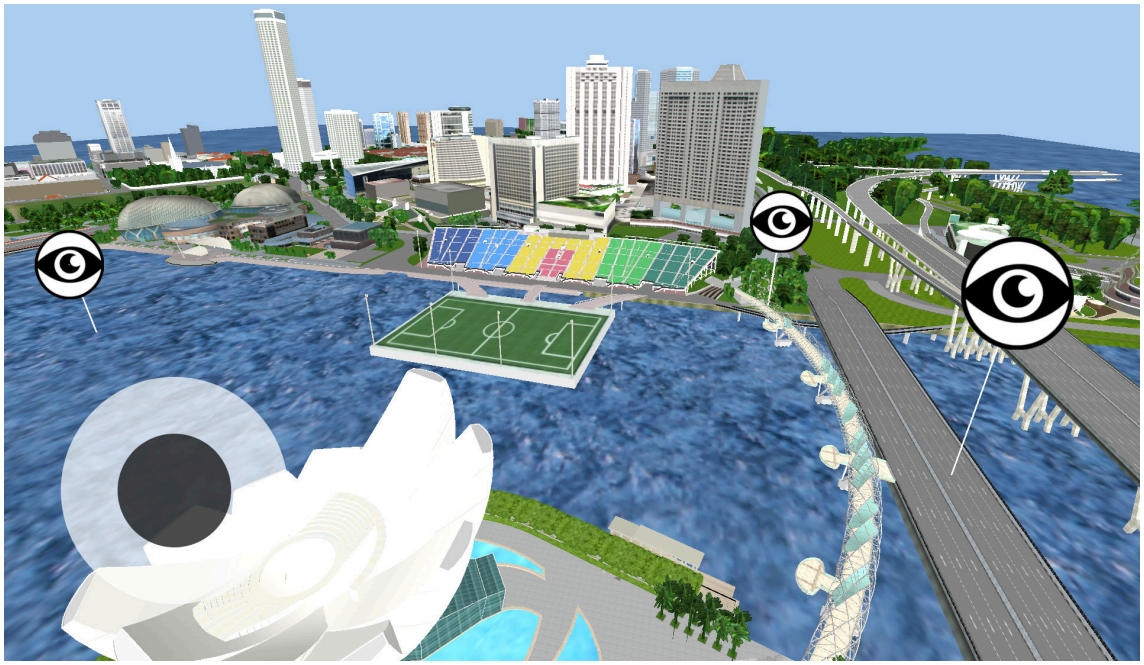


Figure 5.2: Screenshot of the mobile version, with its joystick on the bottom left corner

5.3.1 Bookmark interaction and visual aspect

In Chapter 3 Section 3.2.2, we described two 3D widgets that we use to display bookmarks to users. One of the conclusions of the user-study, described in Section 3.2.3, was that the impact of the way we display bookmark was not significant. In this work, we chose a slightly different way of representing bookmarks due to some concerns with our original representations:

- viewport bookmarks are simple, but non computer vision experts may not be familiar with this type of representation;
- arrow bookmarks are more intuitive to most users, but need to be regenerated when the camera moves, which can harm the rendering framerate.

For these reasons, we changed the bookmarks display to a vertical bar textured with a 2D sprite of a pictorial representation of an eye. The use of such symbol is partly inspired by the cartographic pictograms used to showcase a worthwhile panorama. This 2D sprite is always facing the camera to prevent it from being invisible when the camera would be on the side of it. Screenshots of user interfaces with bookmarks are available in Figures 5.1 and 5.2.

The size of the sprite changes over time following a sine function to help the user distinguish what is part of the scene and what is extra widgets. Since our scene is static, a user knows that a changing object is not part of the scene, but part of the UI.

The other bookmark parameters remain unchanged since Chapter 3: in order to avoid

users to lose context, clicking on a bookmark triggers an automatic, smooth, camera displacement that ends up at the bookmarked camera position. We also display a thumbnail of the bookmark's viewpoint when the mouse hovers a bookmark. Such thumbnail is displayed in Figure 5.1. Note that since on mobile, there is no mouse and thus no pointer, thumbnails are not used in the mobile setting.

Algorithm 6: Computation of the optimal order of segments from a bookmark

```
input  : The bookmarked viewpoint, the ground truth render at this viewpoint,
         the candidate segments
output: The optimal order of the segments
optimal_order ← [];
empty_render ←render(empty_model,bookmarked_viewpoint);
previous_psnr ←psnr(empty_render,ground_truth_render);
total_model ←empty_model;
for i ∈0...200 do
    best_segment ←none;
    best_delta_psnr ←0;
    for segment ∈candidates do
        current_model ←total_model ∪segment;
        current_render ←render(current_model,bookmarked_viewpoint);
        current_psnr ←psnr(current_render,ground_truth_render);
        current_delta_psnr ←current_psnr −previous_psnr;
        if current_delta_psnr >best_delta_psnr then
            best_segment ←segment;
            best_delta_psnr ←current_delta_psnr;
        end
    end
    optimal_order ←optimal_order +best_segment;
    total_model ←total_model ∪best_segment;
end
```

5.3.2 Segments utility at bookmarked viewpoint

Introducing bookmarks is a way to make users navigation more predictable. Indeed, since they are emphasized and, in a way, recommended viewpoints, bookmarks are more likely to be visited by a significant portion of users than any other viewpoint on the scene. As such, bookmarks can be used as a way to optimize streaming by downloading segments in an optimal, precomputed order.

More specifically, segment utility as introduced in Section 4.3.1 is only an approximation of the segment's true contribution to the current viewpoint rendering. When bookmarks are defined, it is possible to obtain a better measure of segment utility by performing an offline rendering at each bookmark's viewpoint. We define $\mathcal{U}^*(s, B_i)$ as being the optimized utility of a segment s in a viewpoint defined at bookmark B_i.

In order to compute the optimized utility of a segment, we developed Algorithm 6, that sorts segments according to their optimized utility. This algorithm takes as input the considered viewpoint, the ground truth rendering from this viewpoint and the set of segments (both geometry and texture) to sort. Starting from an empty model, each segment from the set of candidates is independently added to the scene, and the PSNR between the corresponding render and the ground truth render is computed. We can thus determine which segment brings the highest $\Delta\text{PSNR}/s$, s being the size of the segment in bytes. Once the best segment is found, it is registered, and a new iteration begins. That way, we are able to generate an order of segments sorted by $\Delta\text{PSNR}/s$. This order is then saved as a JSON file that a client can download in order to know which segments contribute the most to a certain viewpoint.

Sorting all the segments from the model would be an excessively time consuming computation. To speed up this algorithm, we only sort the 200 first best segments, and we choose these segments among a filtered set of candidates. To find those candidates, we reuse the ideas developed in Chapter 3. We render the "pixel to geometry segment" and "pixel to texture" maps, as shown in Figure 5.3. These renderings allow us to know which geometry segment and which texture correspond to each pixel, and filter out useless candidates.

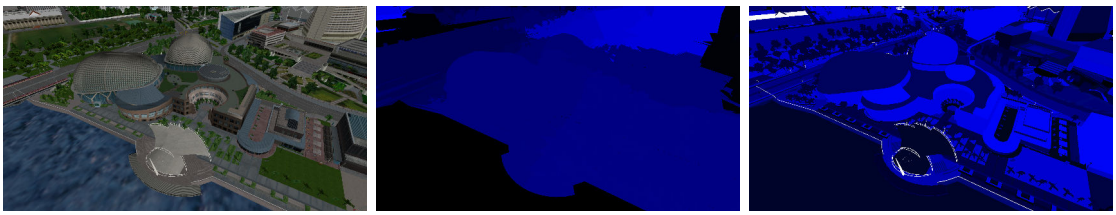


Figure 5.3: A bookmarked viewpoint (left), a pixel to geometry segment map (center), and a pixel to texture map (right)

Figure 5.4 shows how this precomputation improves the quality of rendering. Each curve represents the PSNR one can obtain by downloading a certain amount of data following a streaming policy. The blue curve, labelled "Default order", is obtained by optimizing the utilities as defined in Section 4.3.1, whereas the green curve labelled "Proposed order" uses the sorting computed in Algorithm 6. We can observe that for the same amount of data downloaded, the optimized order reaches a higher PSNR which means that its utility metric is more accurate. Note that this curve is averaged over all the 9 bookmarks of the scene. These bookmarks are chosen to cover the widest area in the scene, and each one faces a particular object-of-interest.

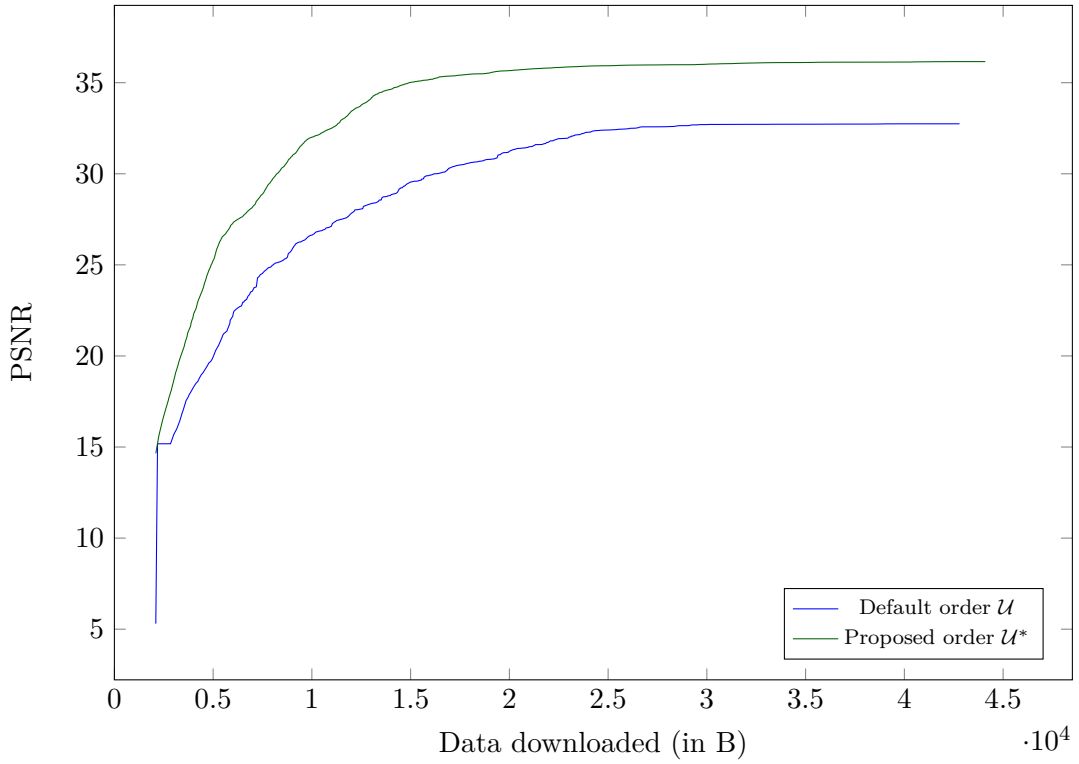


Figure 5.4: Impact of using the precomputed information of bookmarks to select segments to download

5.3.3 MPD modification

We now present how to include bookmarks information in the Media Presentation Description (MPD) file. Bookmarks are fully defined by a position, a direction, and the additional content needed to properly render and use a bookmark in a system. This additional data consist in two files: a thumbnail of the point of view at the bookmark, along with the JSON file giving the optimal segment order for this viewpoint, as computed by Algorithm 6. For this reason, for each bookmark, we create a separate adaptation set in the MPD. The bookmarked viewpoint information is stored as a supplemental property. Bookmarks adaptation set only contain one representation, composed of two segments: the thumbnail used as a preview for the desktop interface and the JSON file.

An example of a bookmark adaptation set is depicted on Snippet 5.1. The three first values in the supplemental property are the camera position coordinates, and the three last values are the target point coordinates.

5.3.4 Loader modifications

We build on the loader introduced in Algorithm 5 to implement a client adaptation logic. We include a bookmark adaptation logic such that (i) when a bookmark is hovered for

```
<AdaptationSet>
    <SupplementalProperty value="156.4909,1.6267,-146.2062,
    157.43106,1.5476,-146.5379" />
 <BaseURL>b1/</BaseURL>
 <Representation>
  <BaseURL>repr1/</BaseURL>
  <SegmentList>
   <SegmentURL media="thumbnail.jpg" />
   <SegmentURL media="order.json" />
  </SegmentList>
 </Representation>
</AdaptationSet>
```

Snippet 5.1: MPD description of a geometry adaptation set, and a texture adaptation set.

the first time, the corresponding thumbnail image as well as the JSON file containing the optimal order of the segments (see Snippet 5.1) are downloaded, and (ii) when a bookmark is clicked, we switch from utility $\mathcal{U}$ to optimized utility $\mathcal{U}^*$ to determine which segments to download next.

Algorithm 7: Algorithm to identify the next segment to query

input : Current index i, time t_i, viewpoint $v(t_i)$, buffer of already downloaded `segments`, MPD, utility metric $\mathcal{U}$, streaming polic Ω
output: Next segment to request, updated buffer

if *bookmark clicking* **then**
  if *not optimal order fetched* **then**
    return optimal order segment;
  else
    return next segment;
  end
else
  `/* Loading policy from previous chapter */`
  (bw_estimation, rtt_estimation) ← `estimate_network_parameters()`;
  candidates ← all_segments
    `.filter(`segment → segment ∉ downloaded_segments`)`
    `.filter(`segment → segment ∈ frustum`)`;
  best_segment ← `argmax(`candidates, segment → $\Omega(\mathcal{U}$, segment`))`;
  downloaded_segments.`append(`best_segment`)`;
  return best_segment;
end

5.4 Evaluation

We now describe our setup and the data we use in our experiments. We then present a user study where users try our new interface with different streaming policies and we compare the impact of the design choices we introduced in the previous sections.

5.4.1 Preliminary user study

Before conducting the user study on mobile devices, we designed a preliminary user study for desktop devices. This experiment was conducted on twelve users, using the 3D model described in the previous chapter (i.e. the Marina Bay district in Singapore). Bookmarks were sampled from the set of locations of user-uploaded panoramic pictures available on Google Maps, and the task consisted in matching real-world pictures to their virtual location on the 3D model: users were presented with an image coming from Google Street View and they were asked to find the exact same location in the 3D model.

Due to the great difficulty of the task was, as well as the relative familiarity of the users with 3D navigation, the user behaviour was biased towards navigating slowly in the scene. The users almost never clicked the bookmarks, much less as they did during the experiment we ran in Chapter 3.

For these reasons, we decided to setup a new experiment, with a less complex task, and we decided to conduct this experiment on mobile device exclusively, to see how bookmarks help people navigate in a scene when controls are more cumbersome.

5.4.2 Mobile navigation user study

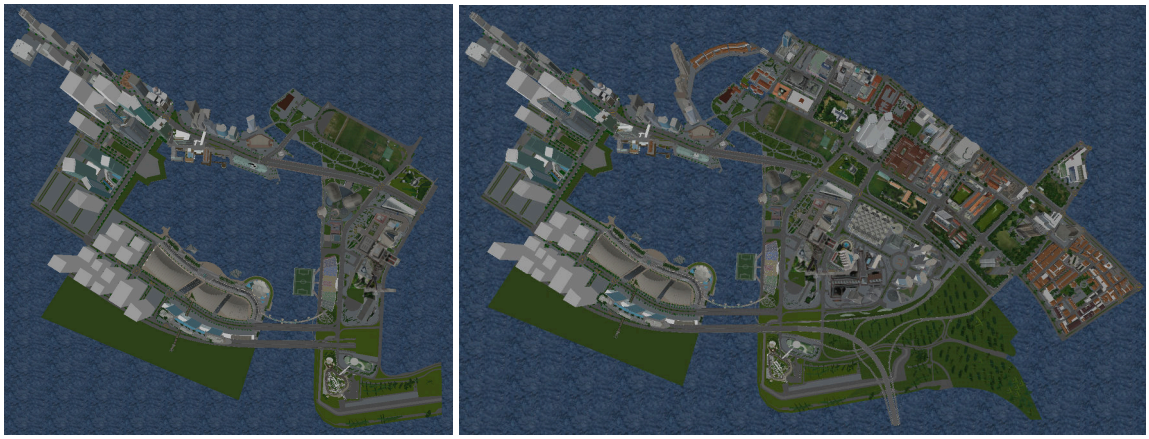
Models

In this user study, we display two successive 3D models to the users:

- For the tutorial phase, we use a model derived from a video game, representing a small scene, in order to prevent users from getting lost in the scene.
- For all the other parts of the experiment, we use a larger version of the Singaporean district 3D model, that include neighbouring districts such as Central Business District. A screenshot of the extended model is given in Figure 5.5 and statistics about sizes are given in Table 5.1.

Experiment

The experiment is articulated into four phases: a tutorial, a comparison between interfaces with and without bookmarks, a comparison between two streaming policies, and a final navigation during which the user is looking for objects in the scene.



(a) Previous chapter model (b) Extended model

Figure 5.5: Models used in our user studies

Files	Previous model	Extended model
OBJ file	62 MB	116 MB
MTL file	0.27MB	0.52 MB
Textures (high res)	167 MB	487 MB
Textures (low res)	11 MB	30 MB

Table 5.1: Sizes of the different files of the model

Tutorial The experiment starts with a tutorial, to get the users accustomed to the controls and the interface. This tutorial showcases the different types of interactions available, including bookmarks, and explains how to use them.

Bookmark The second part of the experiment consists in two 1 minute long sessions: the first session displays a bookmarks-free interface where the only available interactions are translations and rotations of the camera, and the second one augments the interface with bookmarks. There are no special tasks other than to navigate around the model. The part ends with a small questionnaire where users are asked whether they prefer navigating with or without bookmarks, along with a text field to explain their answer.

The main objective of this part of the experiment is not really to know whether people like using the bookmarks or not: we already know from our previous work and from the other parts of this experiment that they do like using the bookmarks. This part most importantly acts as an extended tutorial: the first half trains the users with the basic controls, and the second half trains them to specifically use the bookmarks. This is why we decided not to randomize those two steps at this point.

Streaming This part of the experiment also consists in two 1 minute long sessions that use different streaming policies. One of those experiment has the default greedy policy

described in 4.3.2, and the other one has the enhanced policy for bookmarks described in the previous section. The order of those two sessions is randomized to avoid biases.

Since the behaviours of our streaming policy only differ when the user clicks a bookmark, we design a task where the users have to perform a guided tour of the scene, where each bookmark is a step of the tour. The user starts from anywhere in the scene, and one of the bookmarks is blinking. The user has to touch the bookmark, and observe the recommended viewpoint for a while when arriving at destination. Once some data has been downloaded and the user could get a feeling about the quality of the streaming, another bookmark starts blinking to move one with the tour. This setup is repeated for each streaming policy, and after the two sessions, the users have to answer a questionnaire asking the question *In what session did you find the streaming the smoothest?* The questionnaire also has a text field for users to explain their answer if they wish.

Free navigation The last part of the experiment is a free navigation with an object-finding task. Diamonds are hidden in the scene, and are invisible until the user is close enough. The users have to find the diamonds, and they can navigate by using indifferently the controls and the bookmarks. The loading policy is the default greedy policy for half of the users, and the enhanced policy for bookmarks for the other half, and this order has been randomized.

This is the most important part of the study, as we aim at observing several aspects. First, we hope that users navigate using the bookmarks. Since no guideline has been given to them as to how to interact, we want to observe whether they naturally use the bookmarks or not. In addition, we want to prove the superiority of our bookmark-optimized streaming policy by observing that users tend to perceive a better visual quality (as measured by the PSNR).

Setup

During these experiments, we need a server and a client. The server is hosted on an Acer Aspire V3 with an Intel Core i7 3632QM processor. The user is given a One Plus 5 that is connected to the server via Wi-fi. There is no artificial bandwidth limitation due to the fact that the bandwidth is already limited by the Wi-fi network and by the performances of the mobile device.

Participants

18 users participated in this user-study, 15 males and 3 females, average age is 20.7 and standard deviation is 0.53. We only proposed this user study to relatively young people to ensure they are used to mobile devices: this is a requirement for our user-study since navigating in a 3D scene on a mobile device is hard, and people who are not familiar with

it will likely drop out of the experiment.

5.4.3 Results

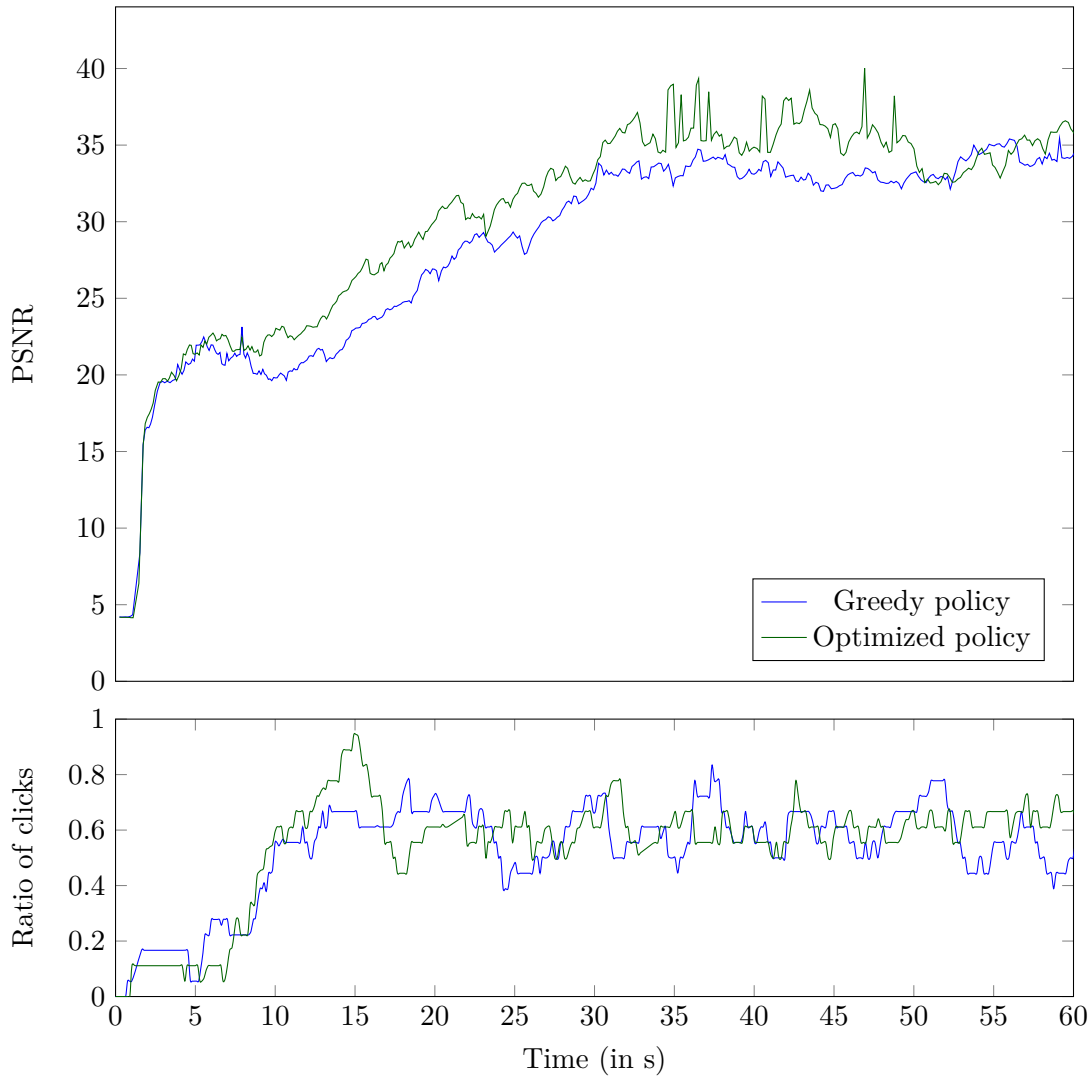


Figure 5.6: Comparison of the PSNR during the second experiment: above, PSNR for greedy and greedy optimized for bookmarks; below, ratio of people clicking on a bookmark.

Qualitative results and observations

We were able to draw several qualitative observations while users were interacting. First, people tend to use and enjoy using the bookmarks, mostly because it helps them navigating in the scene. For the few people who verbalized they do not want to use bookmarks, most often one of the following reasons was invoked:

- they are comfortable enough with using the virtual joystick;
- they find the virtual joystick funnier to use.

We also observe that the gyrocope-based interaction to rotate the camera tends to be either used a lot, either never used: we will not focus on this particular phenomenom as it is out of scope of this study, but it would make an interesting Computer-Human Interaction study.

Quantitative results

Among the 18 participants of this user study, the answers given by users at the end of the **streaming** part of the experiment were as follows: 10 indicated that they preferred the optimized policy, 4 preferred the greedy policy, and 4 did not perceive the difference. One should note that the difference between the two policies can be described in the following terms. The greedy policy tends to favor the largest geometry segments and as a result, the scene structure tends to appear a little bit faster with this method. On the other hand, because it explicitly uses PSNR as an objective function, the optimized policy may result in downloading important textures (that appear large on the screen) before some mid-size geometry segments (that, for example, are typically far from the camera). Some users managed to precisely describe these differences.

Figure 5.6 shows the evolution of the PSNR along time during the second experiment (bookmark guided tour), averaged over all users. Below the PSNR curve is a curve that shows how many users were moving to or staying at a bookmark position at each point in time. As we can see, the two policies have a similar performance at the beginning when very few users have clicked bookmarks. This changes after 10 seconds, when most users have started clicking on bookmarks. A performance gap grows and the optimized policy performs better than the greedy policy. It is natural to observe such performance, as it reflects the fact that the optimized policy makes better decisions in terms of PSNR (as previously shown in Figure 5.4). This probably explains the previous result in which users tend to prefer the optimized policy.

Figure 5.7 shows the PSNR evolution after a click on a bookmark, averaged over all users and all clicks on bookmarks. To compute these curves, we isolated the ten seconds after each click on a bookmark that occurs and we averaged them all. These curves isolate the effect of our optimized policy, and shows the difference a user can feel when clicking on a bookmark.

Figures 5.8 and 5.9 represent the same curves on the third experiment (free navigation). On average, the difference in terms of PSNR is less obvious, and both strategies seem to perform the same way at least in the first 50 seconds of the experiment. The optimized policy performs slightly better than the greedy policy in the end, which can be correlated with a peak in bookmark use occuring around the 50^{th} second. Figure 5.9 also shows

an interesting effect: the optimized policy still performs way better after a click on a bookmark, but the two curves converge to the same PSNR value after 9 seconds. This is largely task-dependent: users are encouraged to observe the scene in experiment 2, while they are encouraged to visit as much of the scene as possible in experiment 3. In average, users therefore tend to stay less long at a bookmarked point of view in the third experiment than in the second.

The most interesting fact is that on the last part of the experiment (the free navigation), the average number of clicks on bookmarks is 3 for users having the greedy policy, and 5.3 for users having the optimized policy. The p-value for statistical significance of this observed difference is 0.06 which is almost low enough to reach the conclusion that a policy optimized for bookmarks could lead users to click on bookmarks more.

Policy	**Number of clicks**									**Average**
Greedy	4	1	1	1	3	3	1	7	6	**3**
Bookmark	3	5	2	5	10	7	6	4	6	**5.33**

Table 5.2: Number of click on bookmarks on the last experiment

Table 5.2 illustrates the number of bookmark clicks for each user (note that distinct users did this experiment on greedy and optimized policy). As we can see, all users clicked at least once on a bookmark in this experiment, regardless of the policy they experiences. However, in the greedy policy setup, 4 users clicked only one bookmark whereas in the optimized policy setup, only one user clicked less than three bookmarks. Everything happens as if users ere encouraged to click on bookmarks with the optimized policy, or that at least some users were discouraged to click on bookmarks with the greedy policy.

5.5 Conclusion

In this chapter, our objective was to propose a mobile interface for DASH-3D and to integrate back the interaction aspects that we developed in Chapter 3. For aesthetics and performance reasons, the UI of the bookmarks has changed, and new interactions were proposed for free navigation in the 3D scene. We developed an algorithm that computes offline a better order of segments for bookmarks than what a greedy policy would do. We encoded this optimal order in a JSON file and we modified our MPD in order to give metadata about bookmarks to the client, as well as modified our client implementation to benefit from this. We then conducted a user study on 18 participants where users had to navigate in scenes with bookmarks and using various streaming policies. The results indicate that users prefer the optimized version of the client streaming policy, which is

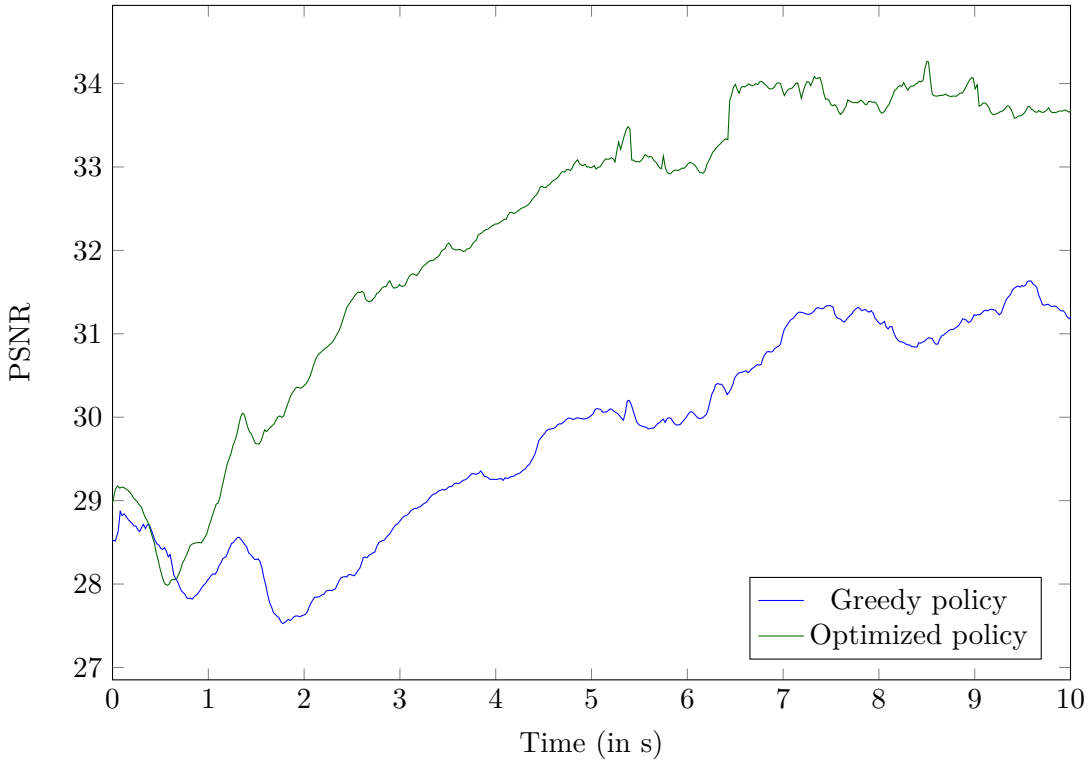


Figure 5.7: Comparison of the PSNR after a click on a bookmark during the second experiment

coherent with the PSNR values that we computed. The results also show that users who enjoy an optimized policy tend to use the bookmarks more.

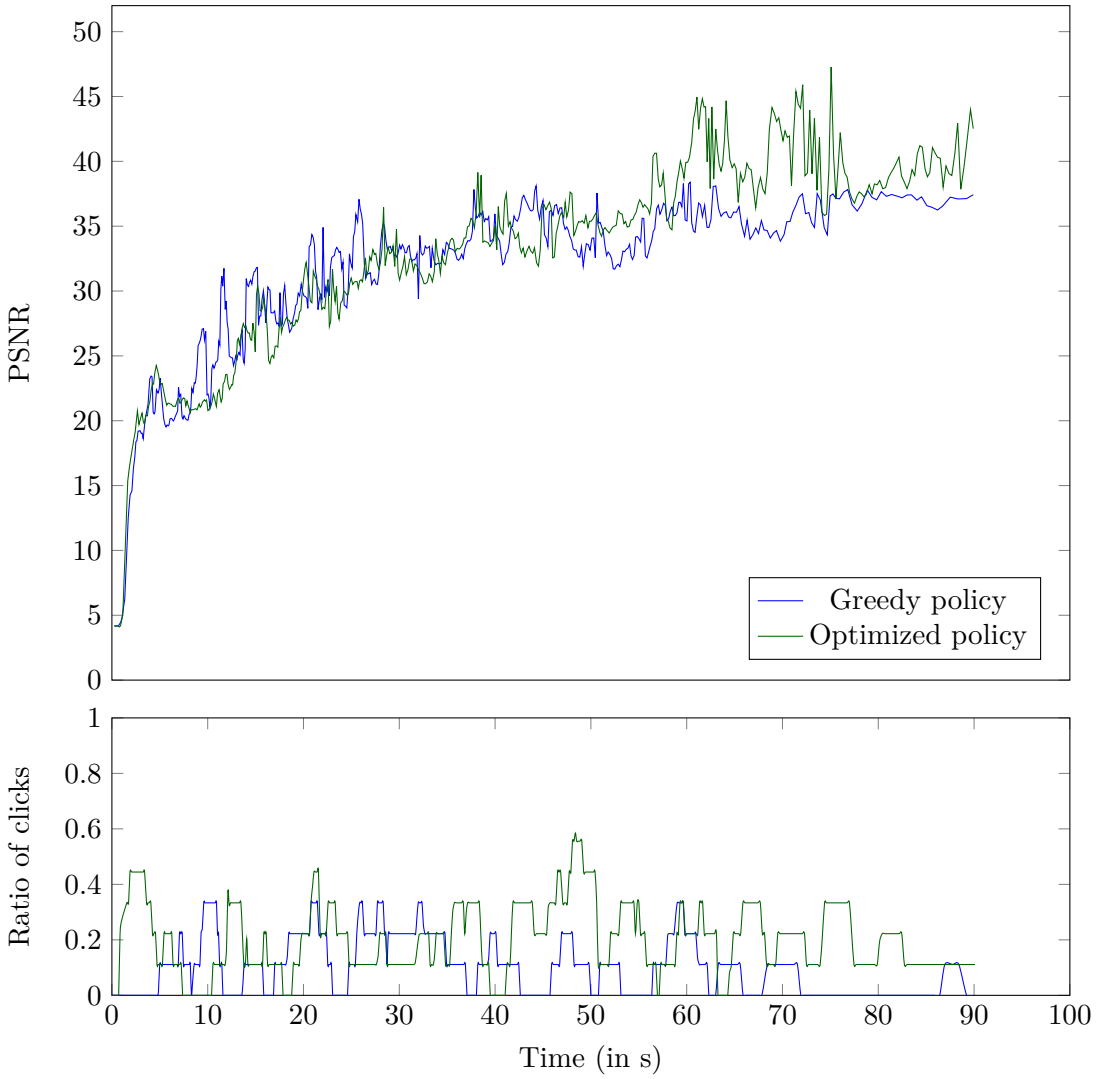


Figure 5.8: Comparison of the PSNR during the third experiment: above, PSNR for greedy and greedy optimized for bookmarks; below, ratio of people clicking on a bookmark.

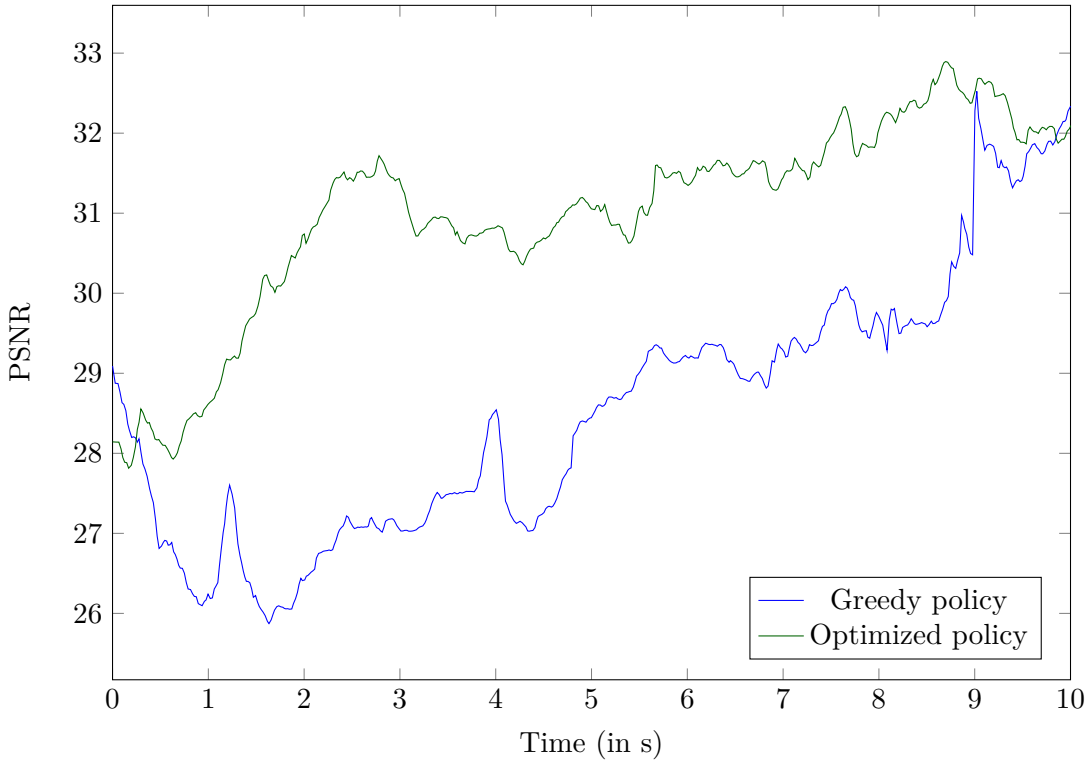


Figure 5.9: Comparison of the PSNR after a click on a bookmark during the third experiment

Conclusion

1 Contributions

In this thesis, we attempted to answer four main problems: **the content preparation**, **the streaming policy and its relation to the user's interaction**, **the evaluation**, and the **implementation**. To answer those problems, we presented three main contributions.

Our first contribution analyzes the links between the streaming policy and the user's interaction. We set up a basic system allowing navigation in a 3D scene (represented as a textured mesh) with the content being streamed through the network from a remote server. We developed a navigation aid in the form of **3D bookmarks**, and we conducted a user study to analyze its impact on navigation and streaming. On one hand, consistently with the state of the art, we observed that navigation aid **helps people navigating in a scene**, since they perform tasks faster and more easily. On the other hand, we showed that benefiting from bookmarks in 3D navigation comes at the cost of a negative impact on the quality of service (QoS): since users navigate faster, they require more data during the same time span. However, we also showed that this cost is not a fatality: using prior knowledge we have about bookmarks, we are able to **precompute an optimal data ordering offline** so that the QoS increases when users click on bookmarks. Simulations on the traces we collected during the user study quantify how these precomputations **improve the QoS**. This work has been published at the ACM MMSys conference in 2016 [Forgione et al., 2016].

After studying the interactive aspect of 3D navigation, we proposed a contribution focusing on the content preparation and the streaming policies of such a system. The objective of this contribution was to introduce a system able to perform **scalable, view-dependent 3D streaming**. This new framework brought many improvements upon the basic system described in our first contribution: support for texture, externalization of necessary computations from the server to the clients, support for multi-resolution textures, rendering performances considerations. We drew massive inspiration from the DASH technology, a standard for video streaming used for its scalability and its adaptability. We exploited

the fact that DASH is made to be content agnostic to fit 3D content into its structure. Following the path set by DASH-SRD, we proposed to tile 3D content using a tree and encode this partition into a description file (MPD) to allow view-dependent streaming, without the need for computation on the server side. On the client side, we implemented loading policies that optimize a utility metric estimating how much geometry and texture segments contribute to the visual rendering of the scene at a particular viewpoint. We thoroughly tested our solutions by running simulations with different parameter values, as well as different loading policies, to propose an efficient framework that we name DASH-3D. This work has been published as a full paper at the conference ACMMM in 2018 [Forgione et al., 2018a]. A demonstration paper on the DASH-3D implementation was also published [Forgione et al., 2018b].

Finally, we brought back the **3D navigation bookmark within our DASH-3D framework**. We developed interfaces that allow navigating in 3D scenes for both **desktop and mobile devices** and we reintroduced bookmarks in these interfaces. The setup of our first contribution considered only geometry, triangle by triangle, which made precomputations and ordering straightforward. Moreover, as the server knew exactly the client needs, it could create chunks adapted to the client's requirements. In DASH-3D, the data are structured a priori (offline), so that chunks are grouped independently of a client's need. We therefore focused on precomputing an optimized order for chunks from each bookmark, and, altered the streaming policies from our second contribution to switch to this optimized order when a user clicks a bookmark. Evaluations showed that the QoS is positively impacted by those policies. A demo paper was published at the conference ACMMM in 2019 [Forgione et al., 2019] showing the interfaces for desktop and mobile clients with bookmarks, but without the streaming aspect. A journal paper will be submitted shortly to value this third contribution.

2 Future work

Successfully adapting the DASH framework to 3D content is a significant step that naturally opens many exciting perspectives. In this section, we shall detail three major perspectives for future work.

2.1 Semantic information

In this thesis, no attention has been given to semantic. Our content preparation considers only spatial information so our adaptation sets and segments may separate data that could be grouped semantically. Having semantic information could help us derive a better structure for our content: we know for example that displaying half a building leads to

poor quality of experience. In order to account for semantic besides partitioning, we could also adapt the utilities we have been defining for our segments: some semantically significant data can be considered as more important than other by taking it into account in our utilities.

2.2 Compression / multi-resolution for geometry

In this thesis, we considered different resolutions for textures, but we have not investigated geometry compression nor multi-resolution. Geometry data are transmitted as OBJ files (mostly consisting in ASCII encoded numbers), which is terrible for transmission. Compression would reduce the size of the geometry files, thus increasing the quality of experience. Supporting multi-resolution geometry would improve it even more, even if performing multi-resolution on a large and heterogeneous scene is difficult. To this day, only little attention has been given to multi-resolution compression for textured geometry [Maglo et al., 2015], and their focus has been on 3D objects. Once again, semantic information could be a great help in this regard. Other compression schemes are also interesting for our framework: [Demir et al., 2016] describes an algorithm to proceduralize architectural models, and the authors also propose a semi-automatic method in [Demir and Aliaga, 2018] to give some control to models editors. Compressing cities through this process can greatly increase the quality of service of our framework.

2.3 Performance optimization

Performance has already been discussed in Chapter 4. However in this thesis, we have for example never discussed removing data from the media engine when it is no longer useful. This means that on a really large scene, performance is bound to become damaged due to the growing amount of data to render, and the saturation of GPU memory. In order for a client (even more a mobile client) to be able to support such scenes, it is necessary to implement a mechanism to periodically free the memory. The utility measures that we described in Section 4.3.1 are good candidates to determine what to unload. We could estimate the performance of our system by measuring variables such as memory used or framerate and use those values to discard data with low enough utility.

Finally, this thesis has proposed a first, but ambitious, complete development of a DASH-3D framework, providing a scalable, efficient, streaming for textured meshes. This framework can be directly used to adapt to semantic or compressed geometry. As the client, especially for mobile clients, special attention is needed for handling locally the large amount of received data for a controlled framerate.

List of Figures

List of Tables

List of Algorithms

List of Snippets

List of Lists

Bibliography

Stack overflow developer survey results. https://insights.stackoverflow.com/survey/2016#technology-most-loved-dreaded-and-wanted, 2016.

Stack overflow developer survey results. https://insights.stackoverflow.com/survey/2017#technology-most-loved-dreaded-and-wanted, 2017.

Stack overflow developer survey results. https://insights.stackoverflow.com/survey/2018#technology-most-loved-dreaded-and-wanted, 2018.

Up to 10x faster 3d tiles streaming. https://cesium.com/blog/2019/05/07/faster-3d-tiles/, 2019.

Stack overflow developer survey results. https://insights.stackoverflow.com/survey/2019#technology-most-loved-dreaded-and-wanted, 2019.

S. Burigat and L. Chittaro. Navigation in 3d virtual environments: Effects of user experience and location-pointing navigation aids. *International Journal of Man-Machine Studies*, 65(11):945–958, 2007.

N. Burtnyk, A. Khan, G. Fitzmaurice, and G. Kurtenbach. Showmotion: camera motion based 3d design review. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 167–174. ACM, 2006.

W. Cheng and W. T. Ooi. Receiver-driven view-dependent streaming of progressive mesh. In *Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 9–14. ACM, 2008a.

W. Cheng and W. T. Ooi. Receiver-driven view-dependent streaming of progressive mesh. In *Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 9–14. ACM, 2008b.

F. Chiariotti, S. D'Aronco, L. Toni, and P. Frossard. Online learning adaptation strategy for dash clients. In *Proceedings of the 7th International Conference on Multimedia Systems*, page 8. ACM, 2016.

L. Chittaro and S. Burigat. 3d location-pointing as a navigation aid in virtual environments. In *Proceedings of the working conference on Advanced visual interfaces, AVI 2004, Gallipoli, Italy, May 25-28, 2004*, pages 267–274, 2004.

L. Chittaro and S. Venkataraman. Navigation aids for multi-floor virtual buildings: A comparative evaluation of two approaches. In *Proceedings of the ACM symposium on Virtual Reality Software and Technology*, pages 227–235. ACM, 2006.

P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Batched multi triangulation. In *VIS 05. IEEE Visualization, 2005.*, pages 207–214. IEEE, 2005.

DASH. Information technology – Dynamic adaptive streaming over HTTP (DASH) – Part 1: Media presentation description and segment formats. Standard, may 2014.

I. Demir and D. G. Aliaga. Guided proceduralization: Optimizing geometry processing and grammar extraction for architectural models. *Computers & Graphics*, 74:257–267, 2018.

I. Demir, D. G. Aliaga, and B. Benes. Proceduralization for editing 3d architectural models. In *2016 Fourth International Conference on 3D Vision (3DV)*, pages 194–202. IEEE, 2016.

J. Eno, S. Gauch, and C. W. Thompson. Linking behavior in a virtual world environment. In *Proceedings of the 15th International Conference on Web 3D Technology*, pages 157–164. ACM, 2010.

C. Erikson, D. Manocha, and W. V. Baxter III. Hlods for faster display of large static and dynamic environments. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 111–120. ACM, 2001.

T. Forgione, A. Carlier, G. Morin, W. T. Ooi, and V. Charvillat. Impact of 3d bookmarks on navigation and streaming in a networked virtual environment. In *Proceedings of the 7th International Conference on Multimedia Systems*, MMSys '16, pages 9:1–9:10, Klagenfurt, Austria, May 2016. ACM. ISBN 978-1-4503-4297-1. doi: 10.1145/2910017.2910607. URL http://doi.acm.org/10.1145/2910017.2910607.

T. Forgione, A. Carlier, G. Morin, W. T. Ooi, V. Charvillat, and P. K. Yadav. Dash for 3d networked virtual environment. In *2018 ACM Multimedia Conference (MM '18), October 22–26, 2018, Seoul, Republic of Korea*, Séoul, South Korea, October 2018a. ISBN 978-1-4503-5665-7/18/10. doi: 10.1145/3240508.3240701.

T. Forgione, A. Carlier, G. Morin, W. T. Ooi, V. Charvillat, and P. K. Yadav. An implementation of a dash client for browsing networked virtual environment. In *2018 ACM Multimedia Conference (MM '18), October 22–26, 2018, Seoul, Republic of Korea*,

Séoul, South Korea, October 2018b. ISBN 978-1-4503-5665-7. doi: 10.1145/3240508.3241398.

T. Forgione, A. Carlier, G. Morin, W. T. Ooi, and V. Charvillat. Using 3d bookmarks for desktop and mobile dash-3d clients. In *2019 ACM Multimedia Conference (MM '19), October 21–27, 2019, Nice, France*, October 2019. ISBN 978-1-4503-9999-9/18/06. doi: 10.1145/1122445.1122456.

D. I. Forum. Guidelines for implementation: DASH-AVC/264 test cases and vectors. http://dashif.org/guidelines/, 2014.

J. Guo, V. Vidal, I. Cheng, A. Basu, A. Baskurt, and G. Lavoue. Subjective and objective visual quality assessment of textured 3d meshes. *ACM Transactions on Applied Perception (TAP)*, 14(2):11, 2017.

M. Guthe and R. Klein. Streaming hlods: an out-of-core viewer for network visualization of huge polygon models. *Computers & Graphics*, 28(1):43–50, 2004.

H. Hoppe. Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 99–108. ACM, 1996.

H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proceedings Visualization'98 (Cat. No. 98CB36276)*, pages 35–42. IEEE, 1998.

T.-Y. Huang, C. Ekanadham, A. J. Berglund, and Z. Li. Hindsight: evaluate video bitrate adaptation at scale. In *Proceedings of the 10th ACM Multimedia Systems Conference*, pages 86–97. ACM, 2019.

J. Jankowski and S. Decker. A dual-mode user interface for accessing 3d content on the world wide web. In *Proceedings of the 21st international conference on World Wide Web*, pages 1047–1056. ACM, 2012.

J. Jankowski and M. Hachet. Advances in interaction with 3d environments. *Comput. Graph. Forum*, 34(1):152–190, 2015.

A. Khan, I. Mordatch, G. Fitzmaurice, J. Matejka, and G. Kurtenbach. Viewcube: A 3d orientation indicator and controller. In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, I3D '08, pages 17–25. ACM, 2008.

G. Lavoué, L. Chevalier, and F. Dupont. Streaming compressed 3d data on the web using javascript and webgl. In *Proceedings of the 18th international conference on 3D web technology*, pages 19–27. ACM, 2013.

F. W. Li, R. W. Lau, D. Kilis, and L. W. Li. Game-on-demand:: An online game engine based on geometry streaming. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 7(3):19, 2011.

K. Liang, R. Zimmermann, and W. T. Ooi. Peer-assisted texture streaming in metaverses. In *Proceedings of the 19th ACM International Conference on Multimedia*, pages 203–212, Scottsdale, AZ, 2011. ACM.

M. Limper, Y. Jung, J. Behr, and M. Alexa. The pop buffer: Rapid progressive clustering by geometry quantization. In *Computer Graphics Forum*, volume 32, pages 197–206. Wiley Online Library, 2013.

P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. L. Faust, and G. Turner. Real-time, continuous level of detail rendering of height fields. Technical report, Georgia Institute of Technology, 1996.

J. D. Mackinlay, S. K. Card, and G. G. Robertson. Rapid controlled movement through a virtual 3d workspace. In *ACM SIGGRAPH Computer Graphics*, volume 24, pages 171–176. ACM, 1990.

A. Maglo, I. Grimstead, and C. Hudelot. Pomar: Compression of progressive oriented meshes accessible randomly. *Computers & Graphics*, 37(6):743–752, 2013.

A. Maglo, G. Lavoué, F. Dupont, and C. Hudelot. 3d mesh compression: Survey, comparisons, and emerging trends. *ACM Computing Surveys (CSUR)*, 47(3):44, 2015.

J. E. Marvie and K. Bouatouch. Remote rendering of massively textured 3d scenes through progressive texture maps. In *The 3rd IASTED conference on Visualisation, Imaging and Image Processing*, volume 2, pages 756–761, 2003.

F. Meng and H. Zha. Streaming transmission of point-sampled geometry based on view-dependent level-of-detail. In *Fourth International Conference on 3-D Digital Imaging and Modeling, 2003. 3DIM 2003. Proceedings.*, pages 466–473. IEEE, 2003.

C. Moerman, D. Marchal, and L. Grisoni. Drag'n go: Simple and fast navigation in virtual environment. In *3D User Interfaces (3DUI), 2012 IEEE Symposium on*, pages 15–18. IEEE, 2012.

O. A. Niamut, E. Thomas, L. D'Acunto, C. Concolato, F. Denoual, and S. Y. Lim. Mpeg dash srd: spatial relationship description. In *Proceedings of the 7th International Conference on Multimedia Systems*, page 5. ACM, 2016.

C. Ozcinar, A. De Abreu, and A. Smolic. Viewport-aware adaptive 360 video streaming using tiles for virtual reality. In *2017 IEEE International Conference on Image Processing (ICIP)*, pages 2174–2178. IEEE, 2017.

C. Portaneri, P. Alliez, M. Hemmer, L. Birklein, and E. Schoemer. Cost-driven framework for progressive compression of textured meshes. In *Proceedings of the 10th ACM Multimedia Systems Conference*, pages 175–188. ACM, 2019.

M. Potenziani, M. Callieri, M. Dellepiane, M. Corsini, F. Ponchio, and R. Scopigno. 3dhop: 3d heritage online presenter. *Computers & Graphics*, 52:129–141, 2015.

S. Rezzonico and D. Thalmann. Browsing 3D bookmarks in BED. In *Proceedings of WebNet 96 - World Conference of the Web Society*, San Francisco, California, USA, October 1996.

F. Robinet and P. Cozzi. Gltf—the runtime asset format for webgl. *OpenGL ES, and OpenGL*, 2013.

R. A. Ruddle, A. Howes, S. J. Payne, and D. M. Jones. The effects of hyperlinks on navigation in virtual environments. *International Journal of Human-Computer Studies*, 53(4):551–581, 2000.

A. Schilling, J. Bolling, and C. Nagel. Using gltf for streaming citygml 3d city models. In *Proceedings of the 21st International Conference on Web3D Technology*, Web3D '16, pages 109–116, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4428-9. doi: 10.1145/2945292.2945312. URL http://doi.acm.org/10.1145/2945292.2945312.

H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. Standard, january 1996.

A. Sideris, E. Markakis, N. Zotos, E. Pallis, and C. Skianis. Mpeg-dash users' qoe: The segment duration effect. In *2015 Seventh International Workshop on Quality of Multimedia Experience (QoMEX)*, pages 1–6. IEEE, 2015.

G. Simon, S. Petrangeli, N. Carr, and V. Swaminathan. Streaming a sequence of textures for adaptive 3d scene delivery. In *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pages 1159–1160. IEEE, 2019.

I. Sodagar. The MPEG-DASH Standard for Multimedia Streaming Over the Internet. *IEEE Multimedia*, 18(4):62–67, apr 2011. ISSN 1070-986X. doi: 10.1109/MMUL.2011.71. URL http://ieeexplore.ieee.org/document/6077864/.

T. Stockhammer. Dynamic adaptive streaming over http –: Standards and design principles. In *Proceedings of the Second Annual ACM Conference on Multimedia Systems*, MMSys '11, pages 133–144, San Jose, CA, USA, Feb 2011. ACM. ISBN 978-1-4503-0518-1. doi: 10.1145/1943552.1943572. URL http://doi.acm.org/10.1145/1943552.1943572.

D. Stohr, A. Frömmgen, A. Rizk, M. Zink, R. Steinmetz, and W. Effelsberg. Where are the sweet spots?: A systematic approach to reproducible dash player comparisons. In *Proceedings of the 25th ACM international conference on Multimedia*, pages 1113–1121. ACM, 2017.

D. Tian and G. AlRegib. Batex3: Bit allocation for progressive transmission of textured 3-d models. *IEEE Transactions on Circuits and Systems for Video Technology*, 18(1): 23–35, 2008.

J. T. Todd. The visual perception of 3d shape. *Trends in cognitive sciences*, 8(3):115–121, 2004.

M. Varvello, S. Ferrari, E. Biersack, and C. Diot. Exploring Second Life. *IEEE/ACM Transactions on Networking (TON)*, 19(1):80–91, 2011.

P. K. Yadav, A. Shafiei, and W. T. Ooi. Quetra: A queuing theory approach to dash rate adaptation. In *Proceedings of the 25th ACM international conference on Multimedia*, pages 1130–1138. ACM, 2017.

S. Yang, C.-H. Lee, and C.-C. J. Kuo. Optimized mesh and texture multiplexing for progressive textured model transmission. In *Proceedings of the 12th Annual ACM International Conference on Multimedia*, MULTIMEDIA '04, pages 676–683, New York, NY, USA, Oct 2004. ACM. ISBN 1-58113-893-8. doi: 10.1145/1027527.1027683. URL http://doi.acm.org/10.1145/1027527.1027683.

M. Zampoglou, K. Kapetanakis, A. Stamoulias, A. G. Malamos, and S. Panagiotakis. Adaptive streaming of complex web 3d scenes based on the mpeg-dash standard. *Multimedia Tools and Applications*, 77(1):125–148, 2018.

R. C. Zeleznik, A. S. Forsberg, and P. S. Strauss. Two pointer input for 3d interaction. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 115–120. ACM, 1997.

Résumé en français

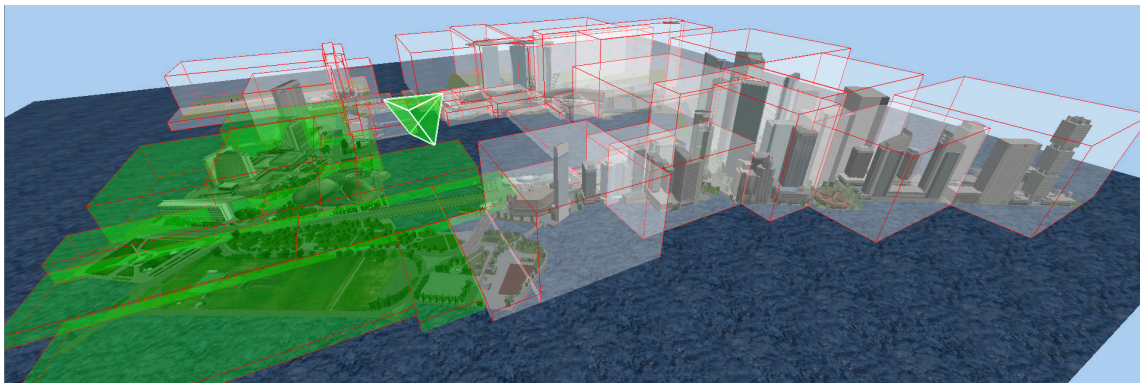


Figure 1: Une scène 3D subdivisée en un arbre k-d avec ses régions délimitées avec des bordures rouges. En blanc, les régions à l'extérieur du champ de vision de la caméra; en vert, les régions à l'intérieur.

Ce chapitre présente la majeure contribution de cette thèse en français, pour les lecteurs non anglophones. Il s'agit de l'adaptation du standard DASH (*Dynamic Adaptive Streaming over HTTP*) pour la transmission vidéo à la transmission de contenu 3D. DASH propose une préparation et une organisation du contenu qui permet l'élaboration de politiques de chargement. Un client DASH est un client qui télécharge la description de l'organisation du contenu (un fichier XML appelé *Media Persentation Description*, MPD), et qui décide, en fonction de ses besoins, ce qui doit être téléchargé. Ces décisions étant prises indépendamment du serveur, celui-ci n'effectue aucun calcul ce qui rend la solution scalable.

Dans ce chapitre, nous montrons comment nous imitons DASH vidéo pour la transmission 3D, et nous développons un système qui hérite des avantages de DASH. La Section 1 décrit la préparation et l'organisation du contenu, ainsi que les métadonnées et les prétraitements effectués sur notre modèle 3D pour optimiser la transmission. La Section 2 propose plusieurs implémentations de clients qui exploitent cette organisation du contenu. La Section 3 évalue l'impact des différents paramètres de notre préparation et de nos clients. Nous concluons enfin dans la Section 4.

1 Formater un NVE en DASH

Dans cette section, nous décrivons comment nous préparons et enregistrons les données 3D de notre NVE (Networked Virtual Environment) dans un format qui soit compatible avec DASH. Dans nos travaux, nous utilisons le format WaveFront OBJ pour les polygones et PNG pour les textures. Cependant, le processus s'applique aussi à d'autres formats.

1.1 Le MPD

Dans DASH, les informations telles que les URL des fichiers, leur résolution ou leur taille sont extraites par le client grâce à un fichier appelé *Media Presentation Description*, MPDLe client ne se base que sur ces informations pour décider quel fichier télécharger et à quel niveau de résolution.

Le MPD est un fichier XML organisé hiérarchiquement en différentes sections. Les *periods* sont le premier niveau, qui dans le cas de la vidéo, indiquent le début et la durée d'un chapitre. Cet élément ne s'applique pas dans le cas d'un NVE, et nous utilisons donc une seule *period* qui contiendra toute la scène, puisque celle-ci est statique.

Chaque *period* contient un ou plusieurs *adaptation sets* qui décrivent des versions alternatives, formats et types de contenu. Nous utilisons les *adaptation sets* pour organiser la géométrie et les textures de la scène.

1.2 *Adaptation sets*

Quand un utilisateur navigue librement dans un NVE, le champ de vision à un moment donné ne contient qu'une partie limitée de la scène. De la même façon que DASH-vidéo partitionne une vidéo en blocs temporels, nous partitionnons les polygones en blocs spatiaux, de sorte que notre client puisse ne télécharger que les blocs nécessaires.

Gestion de la géométrie

Nous utilisons un arbre de partitionnement de l'espace pour organiser les faces en cellules. Une face appartient à une cellule si son barycentre est à l'intérieur de la boîte englobante correspondante. Chaque cellule appartient à un *adaptation set*. Ainsi, l'information géométrique est étalée dans les *adaptation sets* en fonction de leur cohérence spatiale, permettant au client de choisir les faces pertinentes à télécharger. Une cellule est pertinente si son intersection avec le champ de vision de l'utilisateur est non vide. Dans la figure 1, les cellules pertinentes sont représentées en vert.

Puisque notre scène 3D est principalement étalée le long d'un plan horizontal, nous séparons alternativement le modèle dans les deux directions de ce plan.

Nous créons un *adaptation set* séparé pour les grandes faces (par exemple, le ciel ou le sol) puisqu'elles sont essentielles au modèle 3D et qu'elles ne rentrent pas dans les cellules.

Nous considérons une face comme grande si son aire est supérieure à $a + 3\sigma$ où a et σ sont respectivement la moyenne et l'écart-type des aires des faces. Dans notre exemple, ceci correspond aux 5 plus grandes faces, qui représentent 15% de l'aire totale. Nous obtenons ainsi une décomposition du NVE en *adaptation sets* qui partitionne la géométrie de la scène en un *adaptation set* qui contient les faces les plus grandes, et d'autres qui contiennent les faces restantes.

Nous enregistrons la position de chaque *adaptation set*, caractérisée par les coordonnées de sa boîte englobante, dans le MPD comme une propriété supplémentaire de l'*adaptation set*, sous la forme "x_{min}, *largeur,* y_{min}, *hauteur,* z_{min}, *profondeur*" (comme indiqué dans l'extrait de code 1). Ces informations sont utilisées par le client pour implémenter une transmission dépendante de la vue (Section 2).

Gestion des textures

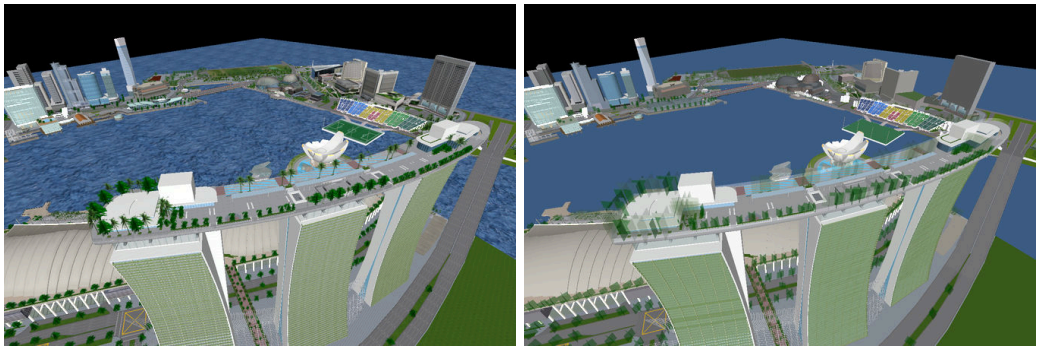
Avec les données de géométrie, nous gérons les textures en utilisant différents *adaptation sets*, indépendants de ceux de la géométrie. Chaque fichier de texture correspond à un *adaptation set* différent, avec différentes *representations* (voir Section 1.3) qui fournissent des résolutions différentes des images. Nous ajoutons à chaque *adaptation set* de texture un attribut qui décrit la couleur moyenne de la texture. Le client peut se servir de cet attribut pour dessiner une face dont la texture n'a pas encore été téléchargée avec une couleur uniforme naturelle (voir Figure 2).

1.3 Représentations

Chaque *adaptation set* peut contenir une ou plusieurs *representations* de la géométrie ou des textures, à différent niveaux de détail (par exemple, avec un nombre différent de faces). Pour la géométrie, la résolution est hétérogène, et appliquer une représentation multi-résolution est pénible : l'aire des faces varie de 0.01 à plus de $10k$, sans tenir compte des faces extrêmes. Pour les scènes texturées, il est commun d'avoir des données hétérogènes puisque l'information peut être enregistrée soit sous forme de géométrie, soit sous forme de texture. Anisi, gérer un compromis entre géométrie et textures est plus adaptable que gérer une combinaison multi-résolution. Pour chaque texture, nous générons des résolutions successives en divisant par 2 la hauteur et la largeur, en s'arrêtant lorsque l'image a une taille inférieure à 64×64. La Figure 2 montre l'utilisation des textures comparée à l'affichage avec une seule couleur par face.

1.4 Segments

Pour permettre l'accès aléatoire au contenu au sein d'un *adaptation set* de géométrie, nous groupons les faces en *segments*. Chaque *segment* est ensuite encodé en un fichier



(a) Avec textures haute résolution (b) Avec les couleurs moyennes

Figure 2: Rendu du modèle avec différents styles de textures

OBJ, qui peut être téléchargé individuellement par le client. Nous partitionnons les faces d'un *adaptation set* en ensembles de N_s faces, en triant les faces par aires décroissantes, et en plaçant ensuite les N_s faces successives dans un *segment*. Ainsi, le premier *segment* contient les faces les plus grandes et le dernier les faces les plus petites. Pour les textures, chaque *representation* contient un *segment* unique.

2 Client DASH-3D

Dans cette section, nous détaillons un client DASH NVE qui exploite la préparation du contenu 3D.

Le client DASH commence par télécharger le MPD, avant de commencer à télécharger les segments. Quand un segment arrive, le client prend la décision du prochain segment à télécharger pour l'afficher quand il arrivera.

Nous considérons une caméra virtuelle qui suit continuellement un chemin $C = \{v(t_i), t_i \in [t_1, t_{final}]\}$, où t_i est l'instant où le segment i est demandé, et au cours duquel les opportunités de téléchargement sont stratégiquement exploitées pour télécharger séquentiellement les segments les plus utiles.

2.1 Utilité des segments

Contrairement au cas de la transmission vidéo, où la taille (en octets) de chaque segment est corrélée à la qualité de la vidéo reçue, pour le contenu 3D, la taille du contenu n'est pas nécessairement corrélée à sa contribution en terme de qualité du rendu. Un grand polygone qui aura un grand impact visuel occupera à peu près autant d'octets qu'un petit polygone. De plus, l'impact visuel dépend du point de vue — un grand polygone lointain ne contribuera pas autant qu'un petit polygone plus près de l'utilisateur. Ainsi, il est important pour un client DASH NVE d'estimer ce que nous appelons *l'utilité d'un*

.

```
<AdaptationSet>
   <SupplementalProperty value="-8834.11230,2201.58853,
      -0.16950, 174.81540,-1344.47740,4767.83367" />
 <BaseURL>as1/</BaseURL>
 <Representation>
  <BaseURL>repr1/</BaseURL>
  <SegmentList>
   <SegmentURL area="2540342.3" size="120K" media="s0.obj" />
   <SegmentURL area="1124.4" size="162K" media="s1.obj" />
   <SegmentURL area="412.6" size="173K" media="s2.obj" />
   <SegmentURL area="270.3" size="147K" media="s3.obj" />
  </SegmentList>
 </Representation>
</AdaptationSet>

<AdaptationSet area="198632.73912" average="178,176,173" mimeType="image/png">
   <BaseURL>textures/MFLOOR07.PNG/</BaseURL>
   <Representation>
      <BaseURL>64x64/</BaseURL>
      <SegmentList>
         <SegmentURL size="7K" mse="57.6" media="t.png" />
      </SegmentList>
   </Representation>
   <Representation>
      <BaseURL>128x128/</BaseURL>
      <SegmentList>
         <SegmentURL size="27K" mse="0.0" media="t.png" />
      </SegmentList>
   </Representation>
</AdaptationSet>
```

Snippet 1: Description d'un *adaptation set* de géométrie, et un de texture, dans le MPD

segment, de sorte à prendre les bonnes décisions de téléchargement.

L'utilité est une fonction d'un segment, qu'il soit de géométrie ou de texture, et du point de vue courant (position de la caméra et direction du regard), et est donc calculé dynamiquement par le client à partir des métadonnées du MPD.

Paramètres statiques

Dans un premier temps, nous allons détailler les paramètres calculés lors de la préparation du contenu, enregistrés dans le MPD.

Tout d'abord, pour chaque segment de géométrie s_G, nous calculons une aire $\mathcal{A}(s^G)$, égale à la somme des aires des polygones du segment. Ensuite, pour chaque segment de texture s^T, le MPD enregistre l'*EQM* (erreur quadratique moyenne) entre l'image courante et l'image à la plus haute résolution disponible. Enfin, pour tous les segments, nous enregistrons la taille du fichier (en octets). En effet, les segments de géométrie ont un nombre similaire de faces, et leurs tailles sont donc à peu près les mêmes. En ce qui concerne les textures, les tailles sont généralement bien plus faibles que celles des segments de géométrie, mais sont aussi très variables, puisqu'entre deux résolutions, le nombre de pixels est multiplié par 4.

Paramètres dynamiques

En plus des paramètres statiques enregistrés dans le MPD pour chaque segment, des paramètres dépendants du point de vue sont calculés pendant la navigation. Premièrement, une mesure d'aire est calculée pour les segments de texture. Puisqu'une texture est peinte sur un ensemble de polygones, on compte pour l'aire de la texture la somme des aires de ces polygones. Nous pourrions calculer cette information de manière statique et l'enregistrer dans le MPD, mais faire ce calcul dynamiquement nous permet de ne prendre en compte que les polygones déjà reçus par le client. Pour une texture T, nous notons l'ensemble des polygones peints par cette texture $\Delta(s^T) = \Delta(T)$ (qui ne dépend que de la texture T et qui est donc constante pour chaque représentation de la texture). À chaque instant t_i, un sous ensemble de $\Delta(T)$ a été téléchargé, nous le notons $\Delta(T, t_i)$.

De plus, chaque segment de géométrie appartient à un *adaptation set* AS^G dont les coordonnées de la boîte englobante sont enregistrées dans le MPD. Étant donné la boîte englobante $\mathcal{BB}(AS^G)$ et le point de vue $v(t_i)$ à l'instant t_i, le client calcule la distance $\mathcal{D}(v(t_i), AS^G)$ à $\mathcal{BB}(AS^G)$ comme la distance du centre de $\mathcal{BB}(AS^G)$ au point principal de la caméra.

Utilité des segments de géométrie

Maintenant, nous avons tous les paramètres pour déduire une mesure d'utilité d'un segment de géométrie. L'utilité des segments de textures est déduite des utilités des segments

de géométrie.

L'utilité d'un segment de géométrie s^G pour un point de vue $v(t_i)$ est

$$\mathcal{U}\Big(s^G, v(t_i)\Big) = \frac{\mathcal{A}(s^G)}{\mathcal{D}(v(t_i), AS^G)^2}$$

où AS^G est l'*adaptation set* qui contient s^G.

Concrètement, l'utilité d'un segment est proportionnelle à l'aire couverte par ce segment, et inversement proportionnelle au carré de la distance entre la caméra et la boîte englobante de son *adaptation set*. De cette manière, nous favorisons les segments contenant des grandes faces, et qui sont proches de la caméra.

Utilité des segments de texture

Pour une texture T, les polygones de $\Delta(T)$ peuvent être dans plusieurs segments de géométrie. Ainsi, pour chaque segment de géométrie déjà téléchargé $s_k^G \in K$, on compte les polygones de $\Delta(T, t_i)$ dans s_k^G, et on calcule ainsi le ratio qu'occupe T dans $\mathcal{A}(s_k^G)$. Nous définissons donc l'utilité, notée $\mathcal{U}\Big(s^T, v(t_i)\Big)$, par

$$psnr(s^T) \sum_{k \in K} \frac{\mathcal{A}(s_k^G \cap \Delta(T, t_i))}{\mathcal{A}(s_k^G)} \mathcal{U}(s_k^G, v(t_i))$$

Concrètement, cette formule définit l'utilité d'un segment de texture grâce à la combinaison linéaire des utilités des segments de géométrie qui utilisent cette texture, pondérées par la proportion occupée par la texture dans le segment. On calcule ensuite un PSNR en utilisant l'erreur quadratique moyenne du MPD et on le note $psnr(s^T)$, de sorte à donner une utilité plus grande à des textures plus haute résolution.

Le client peut ainsi utiliser les utilités définies sur les segments de géométrie et de textures pour sa stratégie de chargement.

2.2 Logique d'adaptation de DASH

Le long du chemin de caméra $C = \{v(t_i)\}$, les points de vue sont indexés par un intervalle de temps continu $t_i \in [t_1, t_{final}]$. Par contraste, la logique d'adaptation de DASH procède séquentiellement le long d'une suite discrète d'instants. La première requête HTTP faite par le client DASH à l'instant t_1 choisit le segment le plus utile s_1^*, qui sera suivi des décisions suivantes aux instants $t_2, t_3, \ldots$. Pour choisir le segment s_i^*, le client doit faire un compromis entre la géométrie et les différentes résolutions de texture en tenant compte du débit, des mouvements de la caméra, et des utilités des segments. La différence entre t_{i+1} et t_i correspond au temps que va mettre s_i^* à être téléchargé. Cette durée peut varier en fonction de la taille du segment et des conditions du réseau. L'algorithme 8 explique comment notre client DASH prend ses décisions.

algorithme 8: Sélection du prochain segment

entrée : Indice courant i, instant t_i, point de vue $v(t_i)$, liste des segments déjà téléchargés $\mathcal{B}_i$, MPD

sorties: Prochain segment s_i^* à télécharger, liste mise à jour $\mathcal{B}_{i+1}$

```
(estimation_bande_passante, estimation_temps_aller_reotur) ←
 estimer_parametres_reseau();
candidats ← segments_mpd
        .filter(meilleur_segment → meilleur_segment ∉ segments_telecharges)
        .filter(meilleur_segment → meilleur_segment ∈ frustum);
meilleur_segment ← argmax(candidats, meilleur_segment →
 Ω (U(meilleur_segment)));
segments_telecharges.append(meilleur_segment);
```

La façon la plus naïve de séquentiellement optimiser $\mathcal{U}$ est de limiter la décision au point de vue courant $v(t_i)$. Dans ce cas, le meilleur segment s à télécharger sera celui qui maximisera $\mathcal{U}(s, v(t_i))$ pour simplement avoir un meilleur rendu au point de vue courant $v(t_i)$. À cause des délais de transmission, ce segment n'arrivera qu'à l'instant $t_{i+1} = t_{i+1}(s)$ qui dépendra des conditions du réseau et de la taille du segment

$$t_{i+1}(s) = t_i + \frac{\texttt{taille}(s)}{\widehat{BW_i}} + \widehat{\tau_i}$$

En conséquence, le segment le plus utile depuis $v(t_i)$ à l'instant t_i sera peut-être moins utile au moment où il arrivera, à l'instant t_{i+1}.

Une meilleure solution est de télécharger un segment qui devrait être plus utile dans le futur. Avec un horizon temporel χ, nous pouvons optimiser le cumul de $\mathcal{U}$ pendant $[t_{i+1}(s), t_i + \chi]$:

$$s_i^* = \underset{s \in \mathcal{S} \setminus \mathcal{B}_i \cap \mathcal{FC}}{\operatorname{argmax}} \int_{t_{i+1}(s)}^{t_i+\chi} \mathcal{U}(s, \hat{v}(t_i))dt \tag{1}$$

Nous avons aussi testé une approche gloutonne qui optimise l'utilité à l'arrivée du segment :

$$s_i^{\texttt{GLOUTON}} = \underset{s \in \mathcal{S} \setminus \mathcal{B}_i \cap \mathcal{FC}}{\operatorname{argmax}} \frac{\mathcal{U}\left(s, \hat{v}\left(t_{i+1}(s)\right)\right)}{t_{i+1}(s) - t_i} \tag{2}$$

3 Évaluation

Nous décrivons maintenant notre installation et les données que nous utilisons dans nos expériences. Nous présentons une évaluation de notre système et une comparaison de l'impact des choix de conception que nous avons introduit dans les sections précédentes.

3.1 Installation expérimentale

Modèle

Dans nos expériences, nous utilisons un modèle du quartier de Marina Bay à Singapour. Le modèle contient 387.551 sommets et 552.118 faces. La géométrie occupe 62 MO et les textures en occupent 167. Nous partitionnons la géométrie dans un arbre k-d jusqu'à ce que les feuilles contiennent moins de 10.000 faces, ce qui nous donne 64 *adaptation sets*, plus un pour les grandes faces.

Navigation des utilisateurs

Pour évaluer notre système, nous avons collecté des traces d'utilisateurs réalistes que nous pouvons rejouer.

Nous avons présenté notre interface web à six utilisateurs, sur laquelle le modèle se chargeait progressivement pendant que l'utilisateur pouvait naviguer. Les interactions disponibles sont inspirées des jeux vidéos à la première personne (le clavier pour se déplacer et la souris pour tourner). Nous avons demandé aux utilisateurs de naviguer et d'explorer la scène jusqu'à ce qu'ils estiment avoir visité les régions les plus importantes. Nous leur avons ensuite demandé d'enregistrer un chemin qui donnerait une bonne présentation de la scène à un utilisateur qui voudrait la découvrir. Toutes les 100 ms, la position et l'angle de la caméra sont enregistrées dans un tableau qui sera ensuite exporté au format JSON. Les traces sauvegardées nous permettent de rejouer chaque enregistrement et d'effectuer les simulations et évaluations de notre système. Nous avons ainsi collecté 13 enregistrements.

Configuration du réseau

Nous avons testé notre implémentation sous trois débits de 2.5 Mbps, 5 Mbps et 10 Mbps avec un temps aller-retour de 76 ms, en suivant les paramètres de [Forum, 2014]. Les valeurs restent constantes pendant toute la durée de la session pour analyser les variations de performance en fonction du débit.

Dans nos expériences, nous créons une caméra virtuelle qui suit un enregistrement, et notre système télécharge les segments en temps réel selon l'algorithme 8. Nous enregistrons dans un fichier JSON les moments où les segments sont téléchargés. En faisant ainsi, nous évitons de gaspiller le temps et les ressources nécessaires à l'évaluation du système pendant que les segments sont en train d'être téléchargés et à l'enregistrement des informations nécessaires pour tracer les courbes présentées dans les prochaines sections.

Machines et logiciels

Les expériences ont été lancées sur un Acer Aspire V3, avec un processeur Intel Core i7 3632QM et une carte graphique NVIDIA GeForce GT 740M. Le client DASH est écrit en

Paramètres	Valeurs
Utilité	Statique, Dynamique, Combinée
Stratégie	Gloutonne, Proposée
Segments	Triés par aire, non triés
Bande passante	2.5 Mbps, 5 Mbps, 10 Mbps

Table 1: Paramètres de nos expériences

Rust, et utilise Glium pour le rendu et reqwest pour le téléchargement des segments.

Métriques

Pour mesurer objectivement la qualité du rendu, nous utilisons le PSNR. La scène rendue a posteriori en utilisant les mêmes chemins mais en ayant téléchargé toute la géométrie et les textures est utilisée comme vérité terrain. Dans notre cas, une erreur de pixel ne peut arriver que lorsqu'une face est manquante ou quand une texture est manquante ou à une résolution trop faible.

Expériences

Nous présentons des expériences qui valident nos choix d'implémentation à chaque étape de notre système. Nous rejouons les chemins créés par les utilisateurs avec différentes conditions de débit tout en variant les composants clés de notre système.

Nous considérons deux stratégies de chargement appliquées à notre client, proposées dans la Section 2. La stratégie gloutonne détermine, à chaque décision, le segment qui maximise l'utilité prédite du segment au moment de son arrivée, ce qui correspond à l'équation (2). La deuxième stratégie de chargement que nous testons est celle proposée dans l'équation (1). Nous avons aussi analysé l'impact du groupement des faces dans les segments de géométrie en fonction de leur aire. Enfin, nous testons différents paramètres de débit pour étudier comment notre système s'adapte à des conditions de réseau différentes.

3.2 Résultats expérimentaux

La Figure 3 montre comment la métrique d'utilité peut exploiter les paramètres statiques et dynamiques. Les expériences utilisent un arbre k-d et la politique de chargement proposée, sur tous les chemins. On observe qu'une métrique d'utilité purement statique donne des mauvais PSNR. Une utilité purement dynamique donne des résultats légèrement meilleurs, notamment grâce a l'élimination des parties à l'extérieur du champ de vision, mais la version combinée décrite dans la Section 2.1 donne les meilleurs résultats.

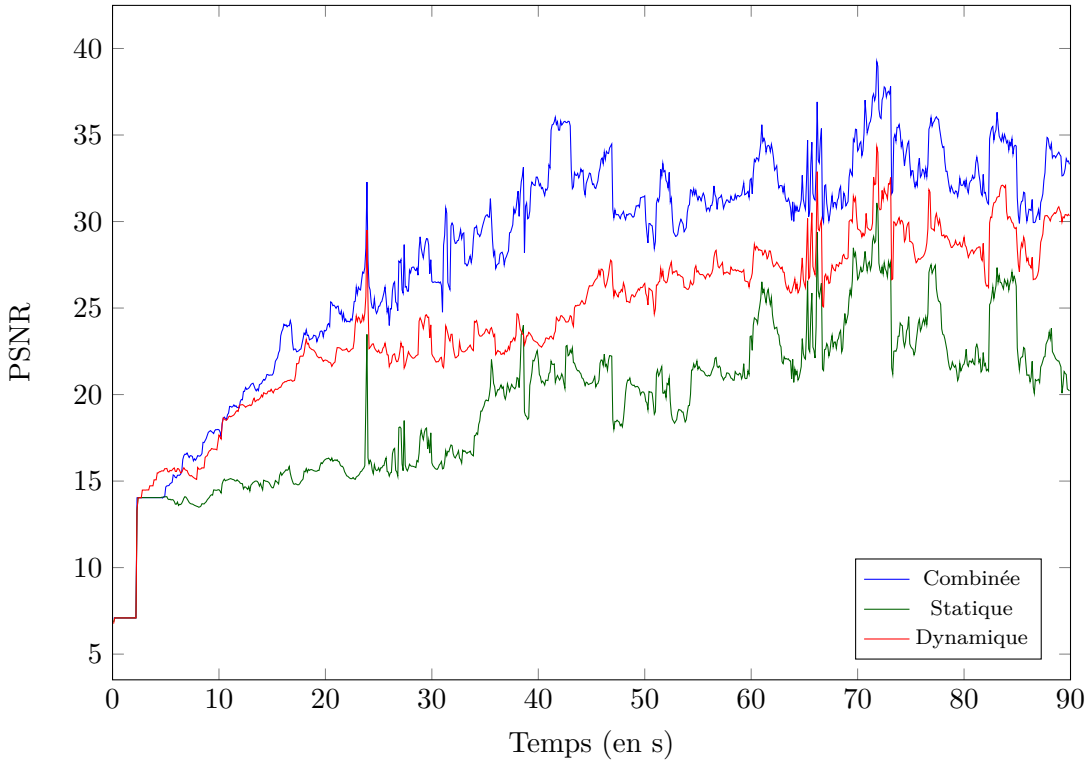


Figure 3: Impact de l'utilité des segments de géométrie sur le rendu à une débit de 5 Mbps.

La Figure 4 montre l'impact de l'arrangement des polygones dans les segments en fonction de leur aire. Il est clair que le PSNR augmente considérablement lorsque l'aire des faces est prise en compte lors de la création des segments. Puisque les segments ont tous la même taille (en octets), trier les faces par aire avant de les ranger dans les segments introduit une asymétrie dans la distribution des aires. Cette asymétrie permet au client de prendre des décisions (télécharger les segments avec la plus grande utilité) et peut créer une grande différence en terme de qualité de rendu.

Nous avons aussi comparé l'approche gloutonne et celle proposée (voir Figure 5) pour un débit limité (5 Mbps). La méthode proposée est meilleure sur les 30 premières secondes et fait mieux en moyenne. La Table 2 montre le PSNR moyen pour les deux méthodes pour différents débits. C'est sur les 30 premières secondes que les décisions sont cruciales puisqu'elles correspondent aux moments où peu de contenu a été téléchargé. Nous observons que notre méthode augmente la qualité du rendu entre 1 et 1.9 dB par rapport à l'approche gloutonne.

La Table 3 montre la distribution des textures téléchargées par les deux approches, à différents débits. La résolution 5 est la plus détaillée, et la résolution 1 la plus grossière. Cette table met en évidence une faiblesse de la politique gloutonne : quand le débit augmente, la distribution des résolutions téléchargées reste plus ou moins la même. En revanche, notre politique s'adapte en téléchargeant des plus hautes résolutions quand le

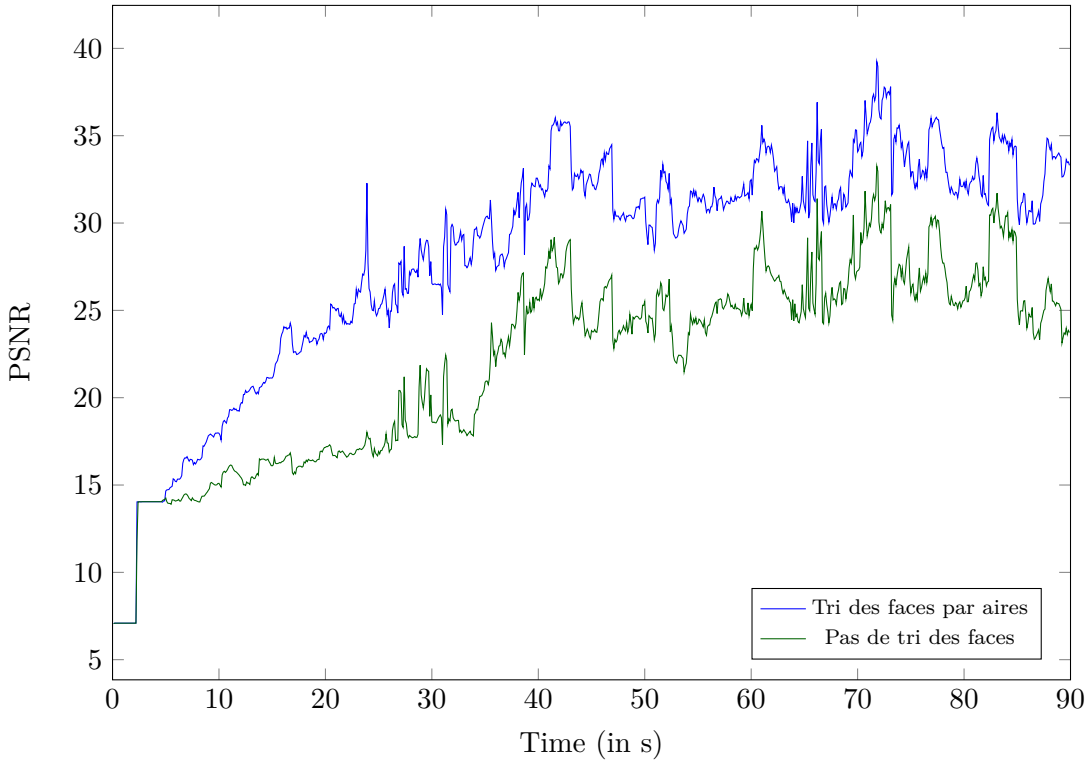


Figure 4: Impact du tri des faces dans les segments à un débit de 5 Mbps

	Premières 30s			**Globalement**		
BP (Mbps)	2.5	5	10	2.5	5	10
Glouton	14.4	19.4	22.1	19.8	26.9	29.7
Proposé	16.3	20.4	23.2	23.8	28.2	31.1

Table 2: PSNR moyens, Glouton vs Proposé

débit est meilleur (13.9% à 10 Mbps contre 0.3% à 2.5 Mbps). En fait, une propriété intéressante de la politique proposée est qu'elle adapte le compromis géométrie-texture au débit. Les textures représentent 57.3% des octets téléchargés à 2.5 Mbps, et 70.2% à 10 Mbps. En d'autres termes, notre système tend à favoriser la géométrie quand le débit est faible, et favoriser les textures quand le débit augmente.

4 Conclusion

Notre travail sur ce chapitre a commencé avec la question : peut-on utiliser DASH pour la transmission de modèles 3D massifs ? La réponse est *oui*. Pour répondre à cette question, nous avons montré comment organiser une soupe de polygones et ses textures dans un format compatible avec DASH qui inclut un minimum de métadonnées utiles au client,

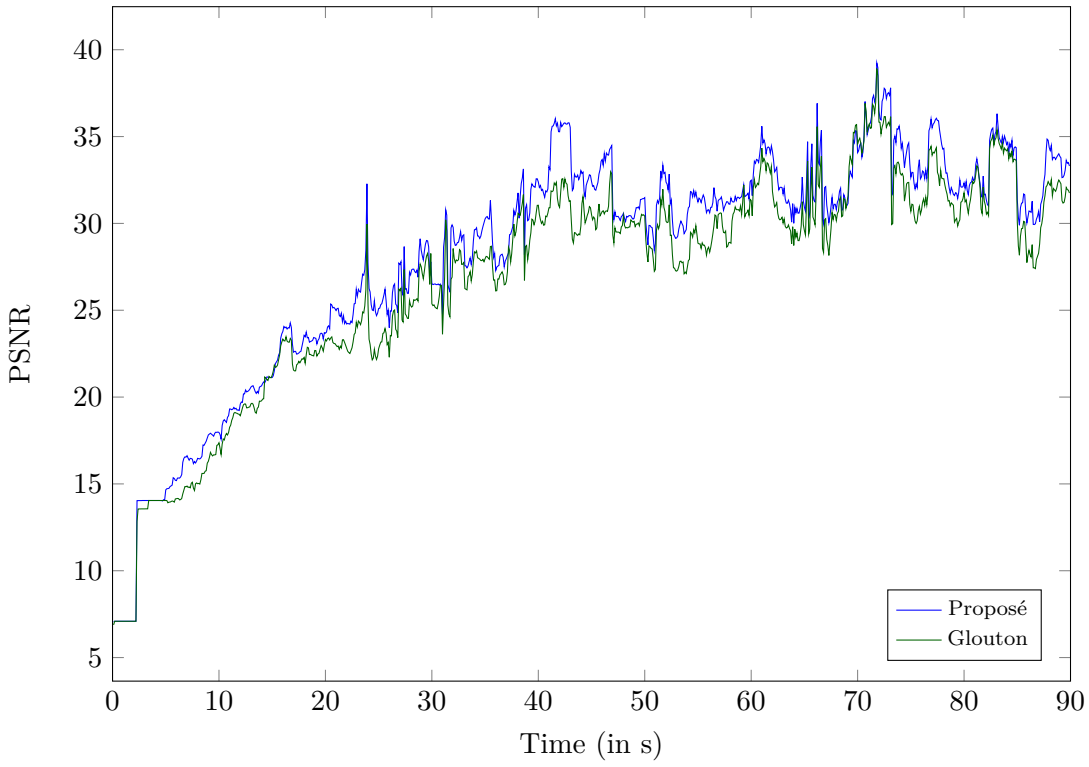


Figure 5: Impact de la politique de chargement (glouton vs proposé) à un débit de 5 Mbps

Résolution	2.5 Mbps	5 Mbps	10 Mbps
1	5.7% vs 1.4%	6.3% vs 1.4%	6.17% vs 1.4%
2	10.9% vs 8.6%	13.3% vs 7.8%	14.0% vs 8.3%
3	15.3% vs 28.6%	20.1% vs 24.3%	20.9% vs 22.5%
4	14.6% vs 18.4%	14.4% vs 25.2%	14.2% vs 24.1%
5	11.4% vs 0.3%	11.1% vs 5.9%	11.5% vs 13.9%

Table 3: Pourcentage d'octets téléchargés pour chaque résolution de texture, pour la politique gloutonne (gauche) et pour celle proposée (droite)

et organise le contenu pour permettre au client de télécharger le contenu le plus utile en premier.

Popularized abstract

More and more 3D models are made available online, and web browsers have now full support for 3D visualization: this thesis focuses on remote 3D virtual environments streaming and interaction, and describes three major contributions.

First, we propose an interface for 3D navigation with bookmarks, which are small virtual objects added to the scene that the user can click to move towards a recommended location. We describe a user study where we analyse the impact of bookmarks on navigation and streaming, and we propose a way to improve the streaming based on the bookmarks.

Secondly, we propose an adaptation of DASH, the video streaming standard, to 3D streaming. We structure the 3D data and textures into chunks, and we propose a client and a few streaming policies that benefit from this structure.

Finally, we integrate our 3D version of DASH and the bookmarks together in a interface for mobile devices, and we describe another study where participants tried this interface.

Résumé vulgarisé

Dans un contexte de démocratisation du nombre et de l'accès à des modèles 3D, nous nous intéressons dans cette thèse à la transmission d'environnements virtuels 3D distants, à travers trois contributions majeures.

Tout d'abord, nous proposons une interface de navigation 3D avec des signets, de petits objets virtuels ajoutés à la scène que l'utilisateur peut cliquer pour se déplacer vers un emplacement recommandé. Nous proposons un moyen d'améliorer la transmission en fonction des signets et évaluons leur impact au travers d'une étude utilisateur.

Ensuite, nous proposons une adaptation de DASH, le standard de la transmission, à la transmission 3D. Nous structurons les données 3D et les textures, et nous proposons un client et des politiques de chargement qui bénéficient de cette structure.

Enfin, nous intégrons notre version 3D de DASH et les signets ensemble dans une interface pour appareils mobiles, et nous décrivons une autre étude où les participants ont essayé cette interface.

Abstract

With the advances in 3D models editing and 3D reconstruction techniques, more and more 3D models are available and their quality is increasing. Furthermore, the support of 3D visualization on the web has become standard during the last years. A major challenge is thus to deliver these remote heavy models and to allow users to visualise and navigate in these virtual environments. This thesis focuses on 3D content streaming and interaction, and proposes three major contributions.

First, **we develop a 3D scene navigation interface with bookmarks** – small virtual objects added to the scene that the user can click on to ease reaching a recommended location. We describe a user study where participants navigate in 3D scenes with and without bookmarks. We show that users navigate (and accomplish a given task) faster when using bookmarks. However, this faster navigation has a drawback on the streaming performance: a user who moves faster in a scene requires higher streaming capabilities in order to enjoy the same quality of service. This drawback can be mitigated using the fact that bookmarks positions are known in advance: by ordering the faces of the 3D model according to their visibility at a bookmark, we optimize the streaming and thus, decrease the latency when users click on bookmarks.

Secondly, **we propose an adaptation of Dynamic Adaptive Streaming over HTTP (DASH), the video streaming standard, to 3D textured meshes streaming**. To do so, we partition the scene into a k-d tree where each cell corresponds to a DASH adaptation set. Each cell is further divided into DASH segments of a fixed number of faces, grouping together faces of similar areas. Each texture is indexed in its own adaptation set, and multiple DASH representations are available for different resolutions of the textures. All the metadata (the cells of the k-d tree, the resolutions of the textures, etc.) is encoded in the Media Presentation Description (MPD): an XML file that DASH uses to index content. Thus, our framework inherits DASH scalability. We then propose clients capable of evaluating the usefulness of each chunk of data depending on their viewpoint, and streaming policies that decide which chunks to download.

Finally, **we investigate the setting of 3D streaming and navigation on mobile devices**. We integrate bookmarks in our 3D version of DASH and propose an improved version of our DASH client that benefits from bookmarks. A user study shows that with our dedicated bookmark streaming policy, bookmarks are more likely to be clicked on, enhancing both users quality of service and quality of experience.

Résumé

Avec les progrès de l'édition de modèles 3D et des techniques de reconstruction 3D, de plus en plus de modèles 3D sont disponibles et leur qualité augmente. De plus, le support de la visualisation 3D sur le web s'est standardisé ces dernières années. Un défi majeur est donc de transmettre des modèles massifs à distance et de permettre aux utilisateurs de visualiser et de naviguer dans ces environnements virtuels. Cette thèse porte sur la transmission et l'interaction de contenus 3D et propose trois contributions majeures.

Tout d'abord, **nous développons une interface de navigation dans une scène 3D avec des signets** – de petits objets virtuels ajoutés à la scène sur lesquels l'utilisateur peut cliquer pour atteindre facilement un emplacement recommandé. Nous décrivons une étude d'utilisateurs où les participants naviguent dans des scènes 3D avec ou sans signets. Nous montrons que les utilisateurs naviguent (et accomplissent une tâche donnée) plus rapidement en utilisant des signets. Cependant, cette navigation plus rapide a un inconvénient sur les performances de la transmission : un utilisateur qui se déplace plus rapidement dans une scène a besoin de capacités de transmission plus élevées afin de bénéficier de la même qualité de service. Cet inconvénient peut être atténué par le fait que les positions des signets sont connues à l'avance : en ordonnant les faces du modèle 3D en fonction de leur visibilité depuis un signet, on optimise la transmission et donc, on diminue la latence lorsque les utilisateurs cliquent sur les signets.

Deuxièmement, **nous proposons une adaptation du standard de transmission DASH (Dynamic Adaptive Streaming over HTTP), très utilisé en vidéo, à la transmission de maillages texturés 3D**. Pour ce faire, nous divisons la scène en un arbre k-d où chaque cellule correspond à un adaptation set DASH. Chaque cellule est en outre divisée en segments DASH d'un nombre fixe de faces, regroupant des faces de surfaces comparables. Chaque texture est indexée dans son propre adaptation set à différentes résolutions. Toutes les métadonnées (les cellules de l'arbre k-d, les résolutions des textures, etc.) sont référencées dans un fichier XML utilisé par DASH pour indexer le contenu: le MPD (Media Presentation Description). Ainsi, notre framework hérite de la scalabilité offerte par DASH. Nous proposons ensuite des algorithmes capables d'évaluer l'utilité de chaque segment de données en fonction du point de vue du client, et des politiques de transmission qui décident des segments à télécharger.

Enfin, **nous étudions la mise en place de la transmission et de la navigation 3D sur les appareils mobiles**. Nous intégrons des signets dans notre version 3D de DASH et proposons une version améliorée de notre client DASH qui bénéficie des signets. Une étude sur les utilisateurs montre qu'avec notre politique de chargement adaptée aux signets, les signets sont plus susceptibles d'être cliqués, ce qui améliore à la fois la qualité de service et la qualité d'expérience des utilisateurs.