國立清華大學電機資訊學院資訊工程研究所

碩士論文

Department of Computer Science
College of Electrical Engineering and Computer Science
National Tsing Hua University

Master Thesis

利用二維圖像資訊及分散渲染系統打造雲端遊戲平台

Building a Next-Generation Cloud Gaming Platform with Planar Map Streaming and Distributed Rendering

王品淳
Pin-Chun Wang

指導教授：徐正炘 博士
Advisor: Cheng-Hsin Hsu, Ph.D.

中華民國 106 年 05 月
May, 2017

國立清華大學
資訊工程研究所

碩士論文

利用二維圖像資訊及分散渲染系統打造雲端遊戲平台

王品淳 撰

106
05

國立清華大學碩士學位論文
口試委員會審定書

利用二維圖像資訊及分散渲染系統打造雲端遊戲平台

Building a Next-Generation Cloud Gaming Platform with Planar Map Streaming and Distributed Rendering

本論文係王品淳君 (105062510) 在國立清華大學資訊工程研究所完成之碩士學位論文，於民國 106 年 05 月 30 日承下列考試委員審查通過及口試及格，特此證明

口試委員：

____

____

____

____

____

所 主 任 ____________________

Acknowledgments

I would like to express my gratitude toward all the people who helped me in the past one and a half years. I would like to thank my adviser: Dr. Cheng Hsin Hsu first, he is the one always giving me guidance and suggestions. Moreover, he gave me chances to strengthen my research skills as well as soft skills and always stand behind me. Secondly, I would like to thank my colleagues in Networking and Multimedia Systems Lab, especially Ching Ling Fan and Hua Jun Hong. Without Ching Ling's careful tutoring, I would have spent more time wandering around and won't be able to graduate in advance. Even though Hua and I are not working at the same project, Hua helps me in many aspects, including programming skills, computer architecture knowledge, and communication skills. Lastly, I would like to thank our colleagues in UIUC, namely Apollo, who even spent a month working in Taiwan. I do enjoy the time accomplishing this work and working with you guys.

TSING HUA UNIVERSITY

中文摘要

本論文為解決現有雲端遊戲平台的限制，提出一個基於二維圖像資訊（planr map）以及分散式的系統架構作為解。我們將製造二維圖像的模組基於伺服器以及客戶端的架構做切割，並加以改良以支援雲端遊戲。本論文更進一步地去優化所需的傳輸頻寬，為此我們為二維圖像資訊設計一壓縮/解壓縮器，由於二維圖像資訊為一自定義的資料型別且和遊戲場景有密切相關，該設計是透過實驗數據作為設計決策。此一設計平台的性能評估顯示其潛力：和x265壓縮器相比，我們的系統（一）在相同頻寬下達到更高的影像畫質、（二）將運算複雜的步驟放置在伺服器端（小於1%），並幾乎達到實時、（三）在高畫質的應用上表現更加優異。本系統可再被加強，如：利用二維圖像資訊的時間相依性提高壓縮率。

Abstract

We propose a new cloud gaming platform to address the limitations of the existing ones. We study the rendering pipeline of 2D planar maps, and convert it into the server and client pipelines. While doing so naturally gives us a distributed rendering platform, compressing 2D planar maps for transmission has never been studied in the literature. In this thesis, we propose a compression component for 2D planar maps with several parametrized modules, where the optimal parameters are identified through real experiments. The resulting cloud gaming platform is evaluated through extensive experiments with diverse game scenes. The evaluation results are promising, compared to the state-of-the-art x265 codec, our platform: (i) achieves better perceptual video quality, by up to 0.14 in SSIM, (ii) runs fast, where the client pipeline takes ≤ 0.83 ms to render each frame, and (iii) scales well for ultra-high-resolution displays, as we observe no bitrate increase when moving from 720p to 1080p, 2K, and 4K displays. The study can be extended in several directions, e.g., we plan to leverage the temporal redundancy of the 2D planar maps, for even better performance.

Contents

口試委員會審定書 **i**

Acknowledgments **ii**

中文摘要 **iii**

Abstract **iv**

1 Introduction **1**
- 1.1 Contributions . . . 3
- 1.2 Thesis Organization . . . 4

2 Background **5**
- 2.1 Cloud Gaming . . . 5
- 2.2 Planar Map . . . 7

3 Using Planar Map in Cloud Gaming **10**
- 3.1 Coordinate Systems . . . 10
- 3.2 Data Format . . . 11
- 3.3 Efficiency of the Two Coordinate Systems . . . 12
- 3.4 System Architecture . . . 12

4 Compressing Planar Maps **14**
- 4.1 Compression Modules . . . 14
- 4.2 Module Parameter Selection . . . 15

5 Evaluations **18**
- 5.1 Implementation . . . 20
- 5.2 Setup . . . 22
- 5.3 Results . . . 23

6 Related Work **29**
- 6.1 Cloud Gaming Platform . . . 29
- 6.2 Mesh Compression . . . 30

7 Discussion **32**
- 7.1 Inter frame compression . . . 32
- 7.2 Integration with Game Engine . . . 32
- 7.3 Holes Filling Issue and User Study . . . 33

8 Conclusion **35**

Bibliography **36**

List of Figures

1.1 Traditional 3D mesh rendering vs. planar map rendering. 2

2.1 General system overview of cloud gaming platforms. 6

2.2 Silhouette illustration. 8

2.3 Project triangle and silhouette on the viewing plane and perform clipping [14]. 9

3.1 Planar map representation in barycentric and Cartesian coordination system. 11

3.2 Sample video frames using different coordinates. 12

3.3 The revised planar map rendering pipeline for our cloud gaming platform. The shaded boxes are from the ordinary planar map rendering pipeline. . 13

4.1 Average video quality with different compression approaches of: (a) delta prediction, (b) entropy coding, and (c) quantization. 15

5.1 Distributed rendering baselines. 18

5.2 Bitrate of our proposed solution at differnt camera,: (a)same R-D curve of 1 basic bunny scene, and (b) all scene under bit-depth 8. 21

5.3 R-D curves of our proposed solution in PSNR compared with x265 in different scene: (a) one basic bunny, (b) one fine-grained bunny, (c) four basic bunny, (d) four fine-grained bunny, (e) eight basic bunny, and (f) eight fine-grained bunny. 25

5.4 Bitrate of our proposed solution at a different bit-depth of 8-bits,: (a)basic bunny scenes, and (b) fine-grained bunny scenes. 26

5.5 Performance improvement of our proposed solution in PSNR with (i) basic bunnies and (b) fine-grained bunnies. 26

5.6 Performance improvement of our proposed solution in SSIM with (i) basic bunnies and (b) fine-grained bunnies. 26

5.7 R-D curves of our proposed solution in PSNR compared with x265 in basic four bunnies scene in: (a) 1080p, (b) 2k, and (c) 4k resolution. . . . 27

5.8 Bitrate required of our proposed solution and x265 with different resolution in: (i) 25 PSNR and (b) 30 PSNR. 27
5.9 Delay categories and its definition. T_0 represents the time client sends the command to sever, where T_1 is the time server receives. Meanwhile, sever transmit encoded planar map at T_2 and client obtained the data at T_3. Finally, the reconstructed scene is played at T_4. 28

7.1 Performance improvement of our proposed solution in PEVQ with (i) basic bunnies and (b) fine-grained bunnies. 33
7.2 One bunny rendered scene in PEVQ scores. 34
7.3 One bunny rendered scene. 34

List of Tables

4.1 Compression results using other performance metrics. 17

5.1 Scenes of test sequences 19
5.2 Running Time (ms), Average/Maximum, 8 Bunnies 24
5.3 Video Quality from Fine-grained Bunnies 24
5.4 Video Quality from Basic Bunnies 25

Chapter 1

Introduction

Cloud gaming refers to: (i) running complex computer games on powerful servers in data centers, (ii) capturing, compressing, and streaming game scenes over the Internet, and (iii) interacting with gamers using thin clients on inexpensive computing devices [39]. In the past few years, we have witnessed strong interests in cloud gaming from both the industrial [36] and academic [19] sides. Existing commercial cloud gaming platforms are *video streaming* based, where the cloud servers perform all the rendering tasks, and the thin clients are merely video decoders. Such a design decision treats computer games as *black boxes* and allows cloud gaming service providers to trade *gaming experience* for *time-to-market*. Video streaming based cloud gaming, however, comes with several limitations, including *high bandwidth consumption, limited scalability, and little room for optimization*, and thus calls for *next-generation* cloud gaming platforms [5].

We make some observations on these three limitations:

- **High bandwidth consumption.** Although video streaming based cloud gaming imposes low computation requirements on thin clients, it incurs high networking bandwidth requirements. For example, PS Now recommends 5 Mbps network bandwidth for good gaming experience [37], which could be increased when the display resolution is even higher. Such a high bandwidth consumption This is partially caused by *under-utilizing* the computing power of thin clients. Nowadays, even inexpensive computing devices, such as low-cost smartphones, come with GPUs (Graphics Processing Units), which are certainly more capable than video decoders. *Moving some rendering tasks from cloud servers to thin clients may reduce the network bandwidth consumption.*
- **Limited scalability.** Since all the rendering tasks are done on cloud servers, supporting more gamers not only leads to higher bandwidth cost, but also results in higher computational cost on cloud gaming service providers. This in turn makes the cloud gaming less profitable and not scalable to many gamers. *Distributing*

rendering tasks among thin clients may improve the scalability.

- **Little room for optimization.** Treating computer games as black boxes prevents cloud gaming platforms from leveraging in-game contexts for performance optimization. *Extracting simplified forms of 3D scenes from computer games may open up a large room for optimization in several aspects, including visual quality, latency, bandwidth consumption, and computational complexity.*

We believe the crucial step of building next-generation cloud gaming platforms is to study the *rendering pipelines* of games for deeper integration between games and platforms, and distribute the rendering tasks among cloud servers and thin clients.

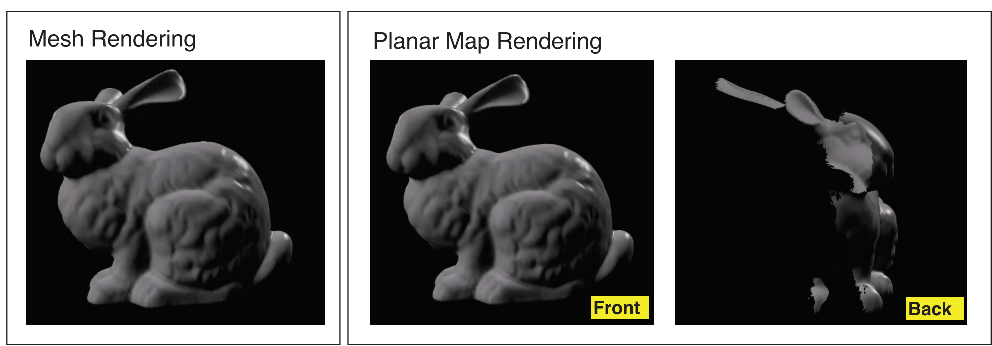


Figure 1.1: Traditional 3D mesh rendering vs. planar map rendering.

[watermark: National Tsing Hua University seal]

We study the rendering pipeline of *planar maps* [2, 3, 14], to understand its potential for addressing the three limitations. The planar map is a vector image consisting of points and edges in the plane, in our case, representing the visible triangles of a 3D scene. Planar maps were first proposed by Baudelaire and Gangnet [3] for graphic design. They define planar maps as 2D graphical objects of arbitrary complexity levels. Planar map tools were then implemented and optimized by Asente et al. [2] for interactive illustration systems. There was however no real-time solution to visibility computation for generation of planar maps until Ellis et al. [14] designed and implemented a planar map pipeline, which is around five times faster than previous work.

Fig. 1.1 illustrates that (i) the 3D mesh rendering pipeline (on the left) renders everything, removing hidden surfaces on the fly via z-buffering, and (ii) the 2D planar map rendering pipeline (on the right) only renders visible triangles, where the depth complexity is reduced to one. Therefore, planar maps are concise and suitable for efficient rendering and streaming. Moreover, because planar maps are vector images, they scale for especially ultra-high-definition displays.

In summary, planar map is an enabler for next-generation cloud gaming platforms because it may: (i) reduce the bandwidth consumption, (ii) increase the scalability, and (iii) help to optimize gaming experience. In this thesis, we apply planar maps in cloud gaming

platforms. This is achieved in two major steps. As the first step, we propose a distributed pipeline for *planar map rendering*. The crux of the new pipeline is the *compressor*, as the compression of planar maps has never been investigated in the literature. In the second step, we design a compressor for planar maps, consisting of several parameterized modules. Using real game scenes, we systematically derive the optimal parameters for a compressor that is specifically designed for *planar map streaming*. The lessons learned in these two steps lead to an efficient end-to-end design of a next-generation cloud gaming platform that leverages planar map streaming and distributed rendering. Our experiment results are quite promising. Compared to video streaming based platforms, our planar map based platform: (i) achieves higher perceptual video quality at the same bitrate, (ii) supports complex scenes when considering perceptual video quality, (iii) runs fast, especially at the client side, and (iv) scales well to ultra-high-resolution displays.

1.1 Contributions

In this thesis, we make the following contributions:

- We study the existing cloud gaming platforms, discover its limitations, and then improve it by proposing a new **distributed rendering client-sever framework**. In particular, we study the proposed real-time planar map rendering pipeline, which generates a low complexity data format and can be re-rendered on thin client. Moreover, we careful define the transmission format, divide the proposed rendering pipeline into client-server framework, and further optimize its performance.
- We study the different representations of a newly invented datatype: planar map in different coordination systems. We evaluate its performance and design its system architecture accordingly. Moreover, we optimize its performance by develop a specific compression pipeline for planar map. The proposed pipeline is testified by real experiments.
- We conduct extensive experiments by recording different sequences varying in model complexity, model numbers, and camera moving speed. We then quantify our proposed system in video quality metrics compared with the state-of-art video transmission solutions in cloud gaming. We also time our components respectively, where the results indicates that we not only achieves realtime computations but also guarantee those computation demanding procedures are placed in the server side, making it possible to support even thin clients.

1.2 Thesis Organization

The rest of this thesis is organized as follows. Chap. 2 provides background on (i) the state-of-art cloud gaming platforms, and (ii) detailed informations towards planar map rendering. This is followed by our proposed system architecture in Chap. 3. We argue our design decision on the system level with real experiment in Chap. 3. Chap. 4 presents our compressor design on a planar map. Chap. 5 evaluates our proposed platform. Chap. 6 surveys the related work, and we discuss the limitations, constraints, and its potential in Chap. 7. Chap. 8 concludes the thesis.

Chapter 2

Background

In this chapter, we survey the existing cloud gaming platform as well as the distributed rendering solutions. Moreover, we study a real-time planar map pipeline while other planar map related literature are listed in Chap. 6.

2.1 Cloud Gaming

Cloud gaming, also known as gaming-on-demand, is a new approach to provide high quality gaming experience to gamers by reducing extra burdens for setting up environments. In cloud gaming, the computation demanding game logics are run in powerful data center (sever side) while the gamers use light-weighted thin client to interact with the games. In this server-client framework, because the gaming scene, often encoded into video, is generated in server side, cloud gaming system naturally support heterogeneous clients and erase the burdens of building powerful clients. In 2000, G-cluster's demonstrate the cloud gaming technology in E3, which offers cloud gaming service over WiFi to hand-held devices. At the early stage, most of the cloud gaming services based on file streaming. More specifically, the gaming service transmits a small part of a game, usually less than 5%, to clients off-lined beforehand so that gamers can start playing quickly. And the remaining game content is downloaded to the thin clients while playing progressively. Even though gamers can start playing without waiting for a long time downloading or buying new disks to update games, the gaming services still requires powerful computation ability at that time.

It is not until 2009, Ross [39] proposed a novel gaming delivery paradigm, cloud gaming is first introduce to academics in literature. Ross' approach renders gaming scene and compress them into video, then transmit them to gamers though the best-effort Internet access. By using Ross' solution, the computation demanding graphic rendering task is offloaded to the cloud, which eliminates the workload on gamer's devices. Yet, a re-

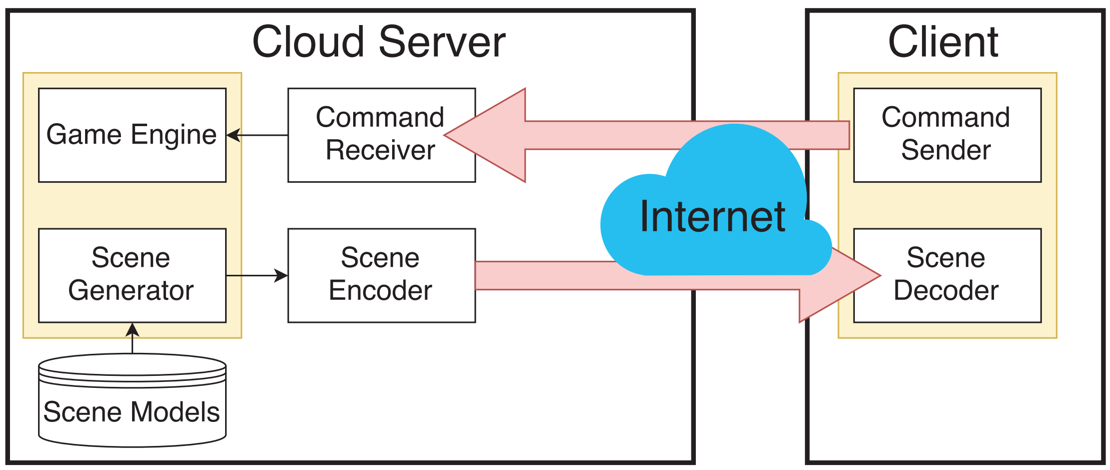


Figure 2.1: General system overview of cloud gaming platforms.

cent literature [28] further generalize the definition of cloud gaming by indicating cloud gaming as a system architecture leverages cloud resources to provide better gaming experience. The cloud gaming system is not restricted to offload rendering task to cloud only, but includes other computation demanding tasks. Fig. 2.1 summarize system architecture cloud gaming generalized from Wei et al. [5]. There are two main components in the cloud gaming server, where game engine handles the gaming logic as well as interact with the user's command and scene generator creates game scene accordingly. On the other hand, the command sender in client side captures commands from gamer's input and receive, decode and display the game scene to gamers. Note that, game scene here could be interpreted in many different data type under different design decisions.

Depending on different transmission data type, cloud gaming platforms can be roughly divided into (i) model based and (ii) image based. For model based method, the server sent vertex, texture, shader information to the clients so that the clients can render a new frame locally. In contrast, image based method process the rendering tasks on the server side, whereas only the result image frame is encoded into videos and transmitted to the client. That is, the image based method requires more bandwidth while the model based method require more computation on the client side. Since there is a trade off between this two method, some literatures propose hybrid method. Cheng and Ooi [7] proposed a receiver driven progressive mesh transmission protocol where the server transmit the base mesh first and the client ask for other content when needed.Besides the transmitting content, the mesh compression algorithm is also proposed by others. Mekuria et al [26] proposed a compression algorithm exploit the properties that the vertex information is location correlated and cut down the required transmission bandwidth. In this thesis, we transmit only the vertex information of visible 2D surface based on gamer's viewpoint, which is called planar map, and reconstruct secne on the client side.

Cloud gaming then brings several benefits to gamers, gaming developers, and service providers. For gamers, cloud gaming supports them to: (i) access to games regardless of places and time, (ii) purchase up-to-date gaming content, (iii) avoid updating their hardware periodically, and (iv) enjoys games across platforms, such as PC, smart phones and mobile devices. For gaming developers, cloud gaming enables them to: (i) reduces the burden of developing games on different platforms and concentrate on game contents, (ii) bypass retailers and open new business model, and (iii) avoid piracy since the gaming content is always on the server side. For service providers, cloud gaming creates demands on cloud resources due to the complexity property of game and brings cloud computing to humans daily life. Because of those tremendous advantages above, companies such as Onlive, Sony's PlayStation (PS NOW) and Nvidia's Grid Gaming Service provides cloud gaming service to customers and increase gamers in the cloud gaming markets.

2.2 Planar Map

Planar map rendering pipelines, such as the one proposed in Ellis et al. [14], have not been customized for distributed rendering in the literature. In this section, we give a high-level overview of the ordinary planar map rendering pipeline. In the next section, we present our refined design of this pipeline for the client-server model. The input of the whole rendering pipeline is the gaming 3D scene and gamer's viewing information. We do require non-penetrating geometry, but do not require particular triangle organization. The planar maps are generated in the following components: (i) silhouette detection,(ii) silhouette clipping, (iii) triangle-triangle occlusion, and (iv) vector rendering.

Silhouette detection. The term silhouette is used loosely here to refer to the visual edge or convex contour edge shared by both a frontfacing and backfacing triangle. Fig. 2.2 demonstrates what is called silhouette, where the face colored in blue is the backfacing and the yellow one is the frontfacing, and $\overline{AB}$ is so called silhouette. We leverage the vertex geometry properties instead of mesh topologies to efficiently detect silhouette. In particular, we use a hash table to record all the edges in the 3D scene as entries. The hash index of each edge is computed as a function of the 3D coordinates of the edge's two vertices. Note that, the hash index is computed by a function of the edge's geometric information so that neighboring triangles are hashed to the same entry when close enough. Once the hash table is constructed, we detect the silhouette edges by checking whether the corresponding face normals have opposite signs in the z component. Lastly, we mark an edge shared by more than two triangles as a silhouette.

Silhouette clipping. We clip each triangle according to those detected silhouettes. First, we remove the non-overlapping triangle-silhouette pairs using their geometric in-

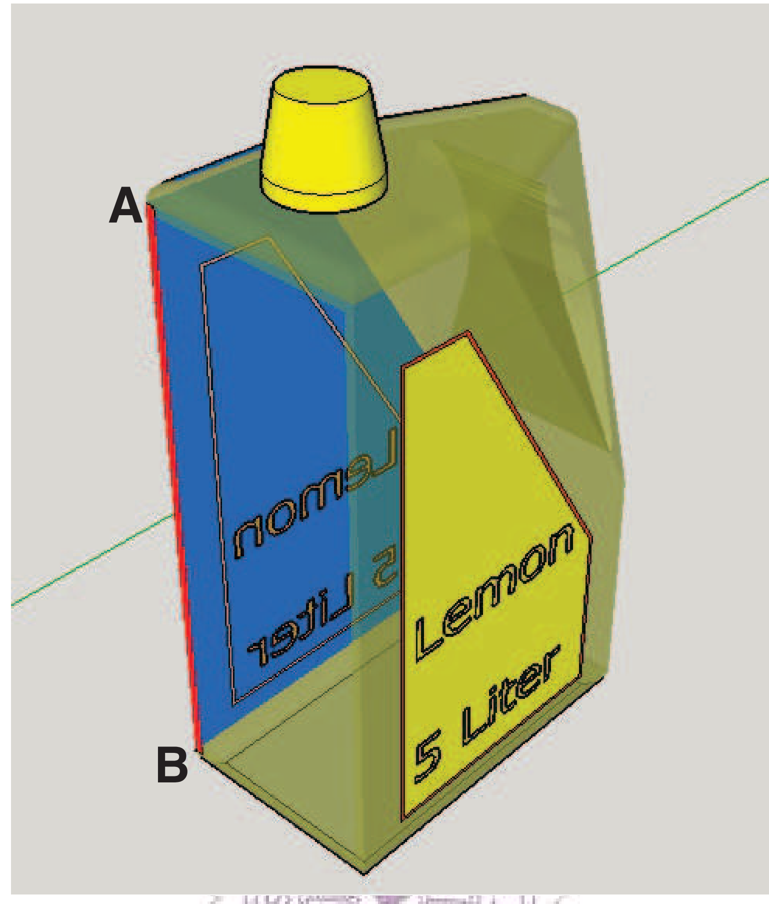


Figure 2.2: Silhouette illustration.

formation. In particular, we project the triangle and silhouette onto the view plane and check whether the silhouette passes through the triangle or not. Secondly, we clip each triangle against its list of overlapping edges. We apply a version of Bernstein and Fussells' clipping algorithm [4] to handle clipping with reduced error. Fig. 2.3 indicates how projecting and clipping is performed. In specific, the algorithm walks around the polygon's vertices using a look up table to determine whether to: (i) output a polygon edge, (ii) generate a new vertex, (ii) output the clipping edge, or (iv) perform some combination of these three actions. In a special case where a huge polygon is behind a complicated 3D object, the polygon is repeatedly clipped. We expect pre-tessellation of such large polygons will alleviate this issue. Lastly, we tessellate all output polygons from this phase back into triangles.

Triangle-triangle occlusion.

After clipping, we obtain all triangles without intersections in z. However, there may be some overlapped triangles that need to be removed. The important property we use to remove occluded triangles is that no triangle is partially occluded. That is, we adopt a lightweight centroid test to determine visibility. We discard the triangles whose centroid

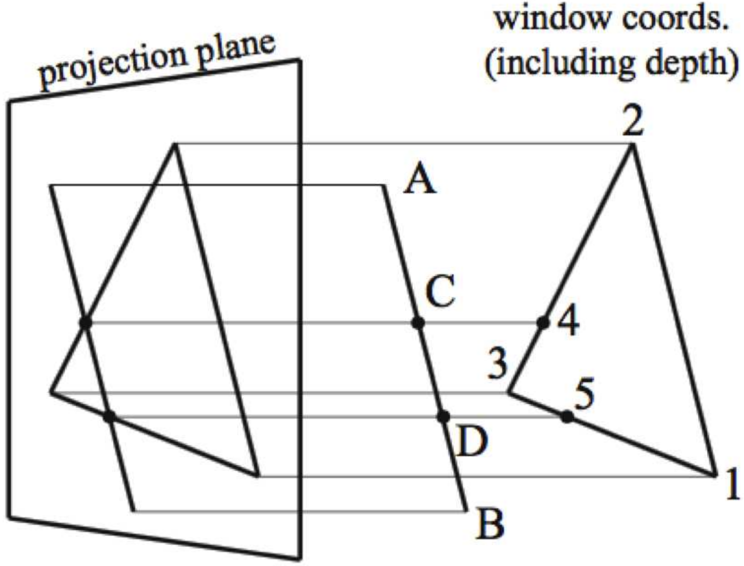


Figure 2.3: Project triangle and silhouette on the viewing plane and perform clipping [14].

is overlapped by any other triangle. With triangle-triangle occlusion, we obtain a set of triangles with the depth complexity of one. This largely simplifies the complexity of the next step: vector rendering.

Vector rendering. The vector renderer works as follows. First, the vector renderer generates the vertex attribute information including 2D position coordinates, texture coordinates, and normal vectors by barycentric interpolation. In fact the barycentric coordinates are key to our implementation. Barycentric coordinates are the input and output of the compressor/decompressor discussed later, and they are the main primitives of transmission between SVGPU and the vector renderer. We thus interpolate vertex attributes from their original positions in the quasi-static database to the clipped positions output by the SVGPU pipeline. In some cases this barycentric interpolation is not necessary, in particular, it suffices to simply assemble the vertex attributes and render them when a triangle has not encountered any clipping in the SVGPU pipeline. We pass the vertex data, as well as the scene information, such as viewing matrix, to the GPU to render the scenes for gamers. Note that the interpolation step is fairly lightweight, and can be implemented in a compute shading language with negligible overhead.

Chapter 3

Using Planar Map in Cloud Gaming

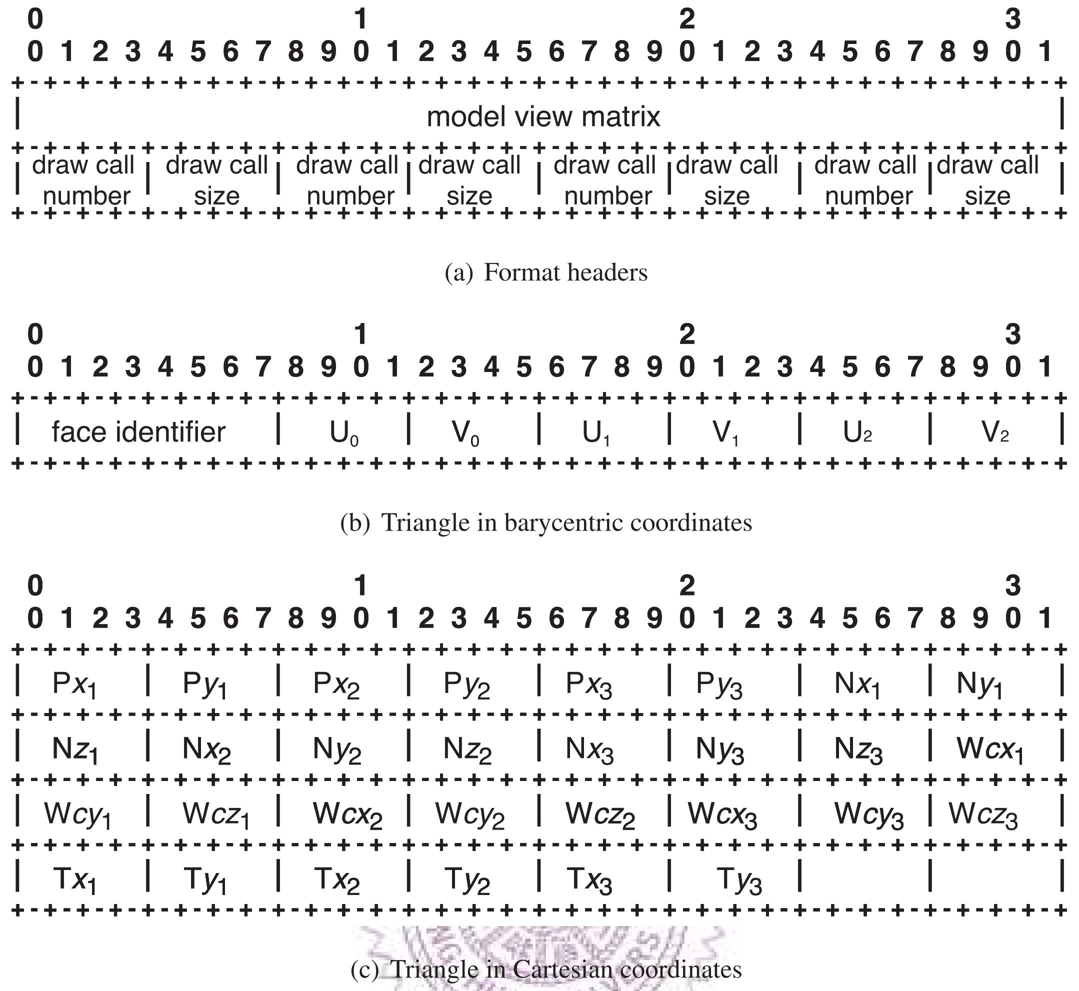
In this chapter, we aim to revised the planar map pipeline in Sec. 2.2 into server-client framework. Eillis et al. [14] proposed planar map to reduce the rendering computation overhead. Yet, this property benefits cloud gaming platforms with the following two reason: (i) cloud gaming platforms have to support thin client, and (ii) reduce transmission bandwidth. In summary, we first look into the planar map format, compared trade-off between different representation, and design a cloud gaming system architecture.

3.1 Coordinate Systems

While the planar maps in the ordinary rendering pipeline [14] are in barycentric coordinate system, the streamed planar maps can be represented using *Cartesian* or *barycentric* coordinate systems. Cartesian coordinates are relative to a single origin, and thus preserve the spatial property across vertices and among video frames. However, Cartesian coordinates do not leverage the (triangles of the) 3D models in the quasi-static database, which may result in unexploited redundancy. In contrast, barycentric coordinates describe each vertex information, including position, normal, and texture, within a triangle (in the quasi-static database) using three floating numbers in $[0, 1]$, say u, v, and $1 - u - v$.

The three coordinates can be seen as the weights of the three vertices of the triangle, e.g., when $u = v = 1/3$, the vertex is the gravity center of the triangle. The merits of barycentric coordinates include: (i) shorter indexes for triangles, and (ii) common triangle patterns on unclipped triangles. However, barycentric coordinates are related to individual triangles, making the correlation among vertices harder to be leveraged. Since the Cartesian and barycentric coordinates both have pros and cons, we consider both coordinate systems.

Through quantitative comparisons in Sec. 3.3, we strive to understand which coordinate system leads to lower bandwidth consumption at the same target gaming quality.



(a) Format headers

(b) Triangle in barycentric coordinates

(c) Triangle in Cartesian coordinates

Figure 3.1: Planar map representation in barycentric and Cartesian coordination system.

3.2 Data Format

We then explain the format of the planar map data sent from the server to the client. This is summarizes in Fig. 3.1. For each video frame, there are three headers: (i) model view matrix, (ii) draw call number, and (iii) draw call size. In particular, the model view matrix *transforms* the world space vertices into model space vertices; the draw call number and size facilitate multi-texture mapping. Moreover, each frame contains a set of *triangles*. With the Cartesian coordinates, each triangle is represented using vertex positions, vertex normals, world space coordinates, and texture coordinates. The vertex positions refer to the 2D positions in the screen space; the vertex numbers and world space coordinates are used for shading; and the texture coordinates are required by adding textures. With the barycentric coordinates, each triangle is represented by a triangle ID (in the quasi-static database), followed by 3 pairs of u, v of individual vertices. In the next section, we present the rendered results and bandwidth consumption comparisons.

3.3 Efficiency of the Two Coordinate Systems

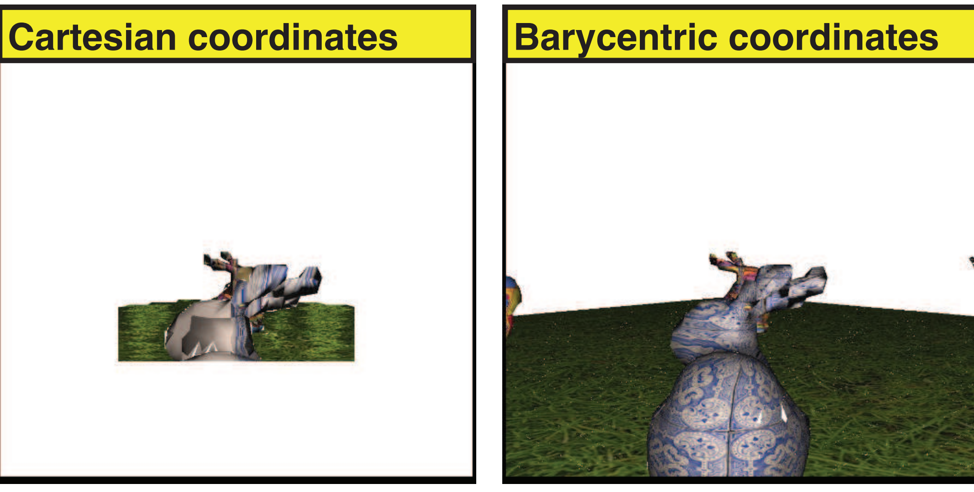


Figure 3.2: Sample video frames using different coordinates.

In Fig. 3.2, we shows the rendered results of using 16-bit quantization on both Cartesian and barycentric coordinates. We can see that the rendered scene from Cartesian coordinates is rather bulky. The reason is that (i) Cartesian coordinates do not have range limits which makes quantization more lossy, (ii) Cartesian coordinates representation required more attributes to represent a triangle, making the quantization entries a lot more. We also observed that there is missing triangle in Fig. 3.2, this is because while reconstructing frames, distortion in normals will leads to failures. Therefore, barycentric rendering uses triangles from models as reference so that the distortion is less than using Cartesian coordinates.

In next section, we design our system architecture using barycentric coordinates.

3.4 System Architecture

To leverage planar maps in cloud games, we divide the ordinary planar map rendering pipeline into the server and client sides. The first three components are put at the server side, and the vector renderer is put on the client side. The design rationale is to have the light weight vector renderer on the client, so that the thin client can render frames *merely* with per-pixel texturing and shading. To connect the server and client pipelines, we add three additional components to our platform: (i) *compressing*, (ii) *decompressing*, and (iii) *streaming sending/receiving*. We also include two quasi-static databases of 3D models (including vertices, texture, and shaders) at the server and the client. The database at the server contains 3D models for all game scenes, while the one at the client stores a subset of 3D models in the current and nearby scenes.

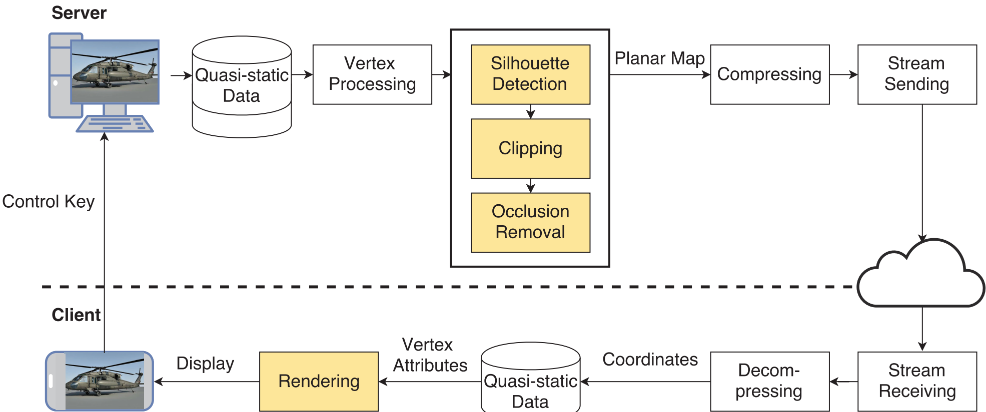


Figure 3.3: The revised planar map rendering pipeline for our cloud gaming platform. The shaded boxes are from the ordinary planar map rendering pipeline.

Fig. 3.3 shows the distributed planar map rendering pipeline, which works as follows. Starting from the server side, game scenes can be viewed as a set of 3D models from the quasi-static database in the model space. With the typical 3D rendering transformations, we can take 3D models from model space, to world space, to view space, and finally into screen space using vertex processing. Then through the silhouette detection, clipping, and occlusion processes (see Sec. 2.2), we obtain 2D planar maps. The 2D planar maps are then compressed to further reduce the required network bandwidth and streamed to client side through the networks. At the client side, we decode the received data stream into coordinates in the decompression component. The quasi-static database at the client side is pre-populated offline, like most computer games. Note that all the quasi-static data are independent of the viewpoints and control inputs from gamers. Instead, they only depend on the game scenes determined by game states. With barycentric coordinate interpolation and the corresponding quasi-static data, we can render the game frames and display them to the gamer by GPU rendering.

The presented rendering pipelines have the heavier components at the resourceful cloud servers, and the lighter component at the thin clients with weak GPUs. Our key optimization problem is the *compression of planar maps* for mitigating high bandwidth consumption, which, to our best knowledge, has not been rigorously studied in the literature. We carefully design and optimize our compressor in the next chapter.

Chapter 4

Compressing Planar Maps

In this chapter, we build compressor/decompressor for planar maps.

4.1 Compression Modules

Quantization.

Quantizing coordinates with enough bits create little visually recognizable difference. Hence, coordinate quantization is considered lossless compression in the mesh streaming literature [30]. Quantization could be classified into uniform and non-uniform quantization, where all quantization regions in uniform quantization are in equal size. Non-uniform quantization, including *scalar* and *vector* quantization, is the foundation of floating-point number compression. For scalar quantization, we apply Lloyd's algorithm [24] on individual dimensions sequentially.

Consider N 2-dimensional inputs (x_n, y_n), where $n = 1, 2, \ldots, N$. The Lloyd's algorithm first produces M optimal regions $g_1^x, g_2^x, \ldots, g_M^x$ for $x_1, x_2, \ldots, x_n$, with representative values $c_1^x, c_2^x, \ldots, c_M^x$. Each x_n falls in one of the regions g_m^x, and is then quantized to c_m^x. The algorithm is then repeated on $y_1, y_2, \ldots, y_N$. We write the scalar-quantized coordinates of (x_n, y_n) as $(q_s(x_n), q_s(y_n))$, for all $n = 1, 2, \ldots, N$. For vector quantization, *all dimensions* are jointly quantized using K-means algorithm [1], in which all inputs are clustered in M several groups, and the centroid of each group is determined.

Delta prediction. To leverage the properties that close-by vertices share similar information, a prediction algorithm may use previous coordinates to predict current coordinates [34]. The prediction step is crucial for high compression ratio [34]. We adopt the delta prediction [12] approach. Consider N 2-dimensional inputs (x_n, y_n). Only the first input (x_1, y_1) is encoded into a symbol, say as a 32 bit floating number. Others are represented in the deltas compared to the previous input, that is $(\Delta x_n, \Delta y_n) = (x_n - x_{n-1}, y_n - y_{n-1})$. $(\Delta x_n, \Delta y_n)$ is then encoded with variable length of code accord-

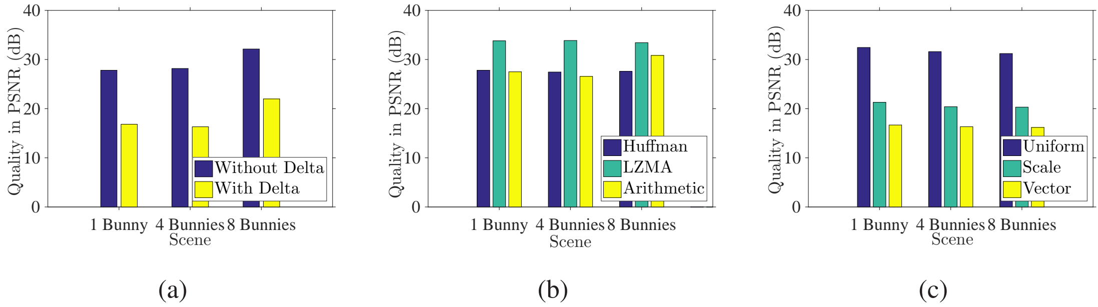


Figure 4.1: Average video quality with different compression approaches of: (a) delta prediction, (b) entropy coding, and (c) quantization.

ing to the histogram (details giving below).

Entropy coding. Entropy coding further exploits the different symbol frequency to reduce transmission bandwidth. We choose two widely used entropy codecs: *Huffman* and *arithmetic* coding [11]. Considering N pairs of distinct symbols along with its probability (x_n, p_n), where $n = 1, 2, \ldots, N$. The Huffman coding builds a Huffman tree with the more frequent elements at lower levels of the tree. We then assign shorter codes to the symbols closer to the root. We denote Huffman coded sequence x_n as $e_h(x_n)$. For arithmetic coding, it converts the whole symbol sequence into a floating point between 1 and 0. The procedure loops through the symbols and shrinks the interval based on the symbol probability. More precisely, the symbols with higher frequency shrinks less. We write the arithmetic encoded sequences of x_n as $e_a(x_n)$. In addition, we also consider the Lempel Ziv Markov Chain Algorithm (LZMA) and use 7-zip as the implementation. LZMA is a lossless dictionary compressor, which encodes a stream with an adaptive binary range coder. We notice that the coordinates may not be byte-aligned, which may be difficult for the entropy coders to handle. We therefore expand the symbols to the next byte boundary before entropy coding, e.g., 7-bit symbols are padded with one extra highest zero bit. Some sanity checks show that the padding strategy reduces the bitrate with no penalty on video quality.

4.2 Module Parameter Selection

We record three game scenes in 720p resolution, in which we vary the number of the popular Bunny model, among 1, 2, and 8. Details in Tab. 5.1 rows 1, 3, and 5. In every scene, bunnies with different textures and standing positions are placed on a plane, and the viewing position and orientation changes according to the gamer's speed. Each scene lasts for 10 secs at 30 Hz frame rate. We consider the following performance metrics to quantitatively compare module parameters.

- **Video quality.** The rendered quality in PSNR, SSIM, and Perceptual Evaluation of Video Quality (PEVQ). PEVQ is a video quality metric described in ITU-T J.247 Annex B [20], and implemented in Skarseth et al. [40]. PEVQ scores are between 0–5, where 0 is the highest score.
- **Compression ratio.** The compressed stream size over the uncompressed stream size.

We have sent the game scenes through a compressor with different quantization, delta prediction, and entropy coding [34] algorithms. We report four most important findings below. Note that for comparison, we use vary the compression depth in quantization and use linear interpolation to round the PSNR score under different modules.

Delta prediction negatively impacts compression ratio as well as video quality. We measure the PSNR and compressed stream size with and without delta prediction. Fig. 4.1(a) plots the achieved average video quality of different scenes with and without the delta prediction at a bitrate of 2.3 Mbps. We find that the delta prediction makes more variance on the resulting symbols, which then leads to lower compression ratio. Moreover, because barycentric coordinates are all positive numbers, applying delta prediction requires one extra sign bit, which worsens the performance.

LZMA outperforms other entropy coding algorithms. We compress each scene multiple times with different entropy coders and plot the average PSNR at the same compression ratio using these entropy coders in Fig. 4.1(b). It is clear that LZMA outperforms the other two entropy coders.

Uniform quantization outperforms other quantization approaches. On the 2D plane, the vector quantization divides the space into quadrilaterals, and the scale quantization divides the space into variable-size rectangles. In contrast, uniform quantization crops the space into equal size blocks. We plot the average PSNR from the three quantization approaches at the same compression ratio in Fig. 4.1(c). This figure reveals that the uniform quantization outperforms others by far. We believe this is because the barycentric coordinates are between 0 and 1, and have weak clustering property.

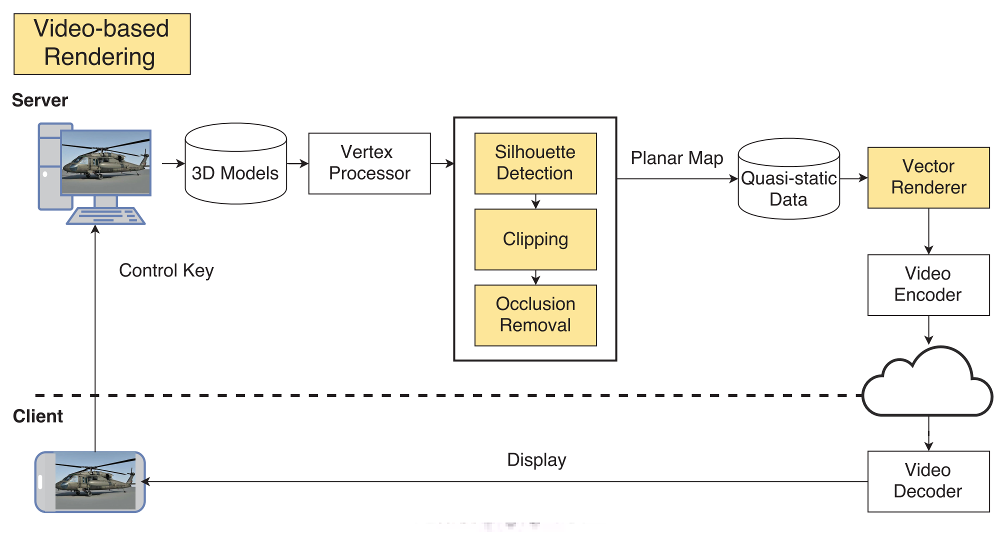
Based on the above findings, we adopt *barycentric coordinates, uniform quantization, and LZMA coder* to compress the 2D planar maps. We note that while we report sample results in PSNR, results in SSIM as summarized in Tab. 4.1, also support the same findings.

Table 4.1: Compression results using other performance metrics.

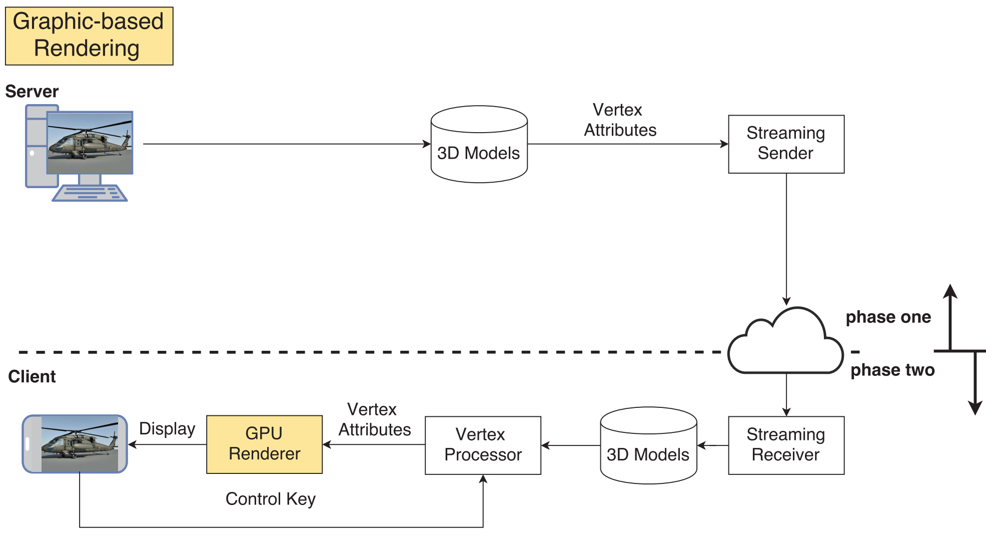
Delta Prediction: with/without Delta Prediction			
	1 Bunny	4 Bunnies	8 Bunnies
SSIM	0.87/0.94	0.88/0.94	0.85/0.94
Quantization: Scale/Uniform/Vector			
	1 Bunny	4 Bunnies	8 Bunnies
SSIM	0.93/0.98/0.88	0.86/0.97/0.87	0.86/0.97/0.86
Entropy Coding: Huffman/LZMA/Arithmetic			
	1 Bunny	4 Bunnies	8 Bunnies
SSIM	0.94/0.98/0.94	0.93/0.98/0.92	0.94/0.98/0.96

Chapter 5

Evaluations



(a) video-based rendering platform



(b) graphic-based rendering platform

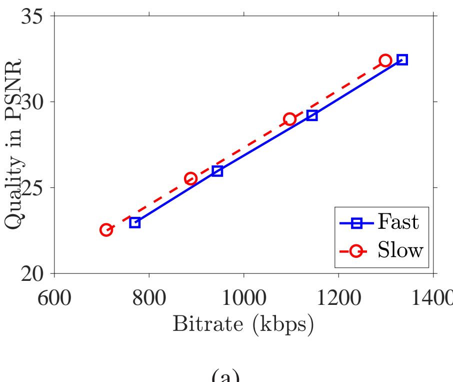
Figure 5.1: Distributed rendering baselines.

Table 5.1: Scenes of test sequences

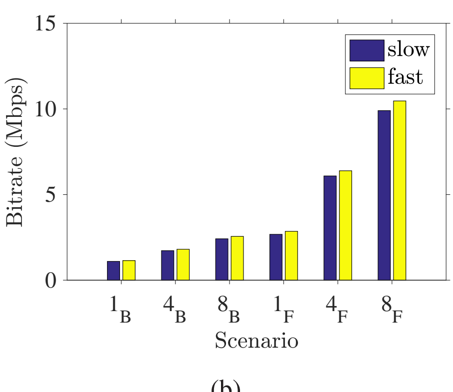
Scene at T	Scene at $T + 100$	Description
		• One bunny • Fast viewpoint moving speed • Basic model
		• One bunny • Slow viewpoint moving speed • Basic model
		• Four bunnies • Fast viewpoint moving speed • Basic model
		• Four bunnies • Slow viewpoint moving speed • Basic model
		• Eight bunnies • Fast viewpoint moving speed • Basic model
		• Eight bunnies • Slow viewpoint moving speed • Fine-grained model

5.1 Implementation

In this section, we describe how we implement each component respectively.



(a)



(b)

Figure 5.2: Bitrate of our proposed solution at differnt camera,: (a)same R-D curve of 1 basic bunny scene, and (b) all scene under bit-depth 8.

SVGPU. We modified planar map rendering pipeline [14], where vertex attributes interpolation has been implemented in CUDA, C++, and OpenGL, to handle more complex scenes and enable it for rendering continuous sequences. In previous pipeline, there is less model overlap issues so that guard band clipping is then implemented to overcome complex model overlapping scene. Also, we adjust barycentric coordinates to account for perspective correction. Moreover, we tune the planar map rendering pipeline for using gamer's view point and viewing directions as input. We then write different bash scripts to simulation gamer's movement and record the generated planar map as well as the output frames as our test scenes, detailed in Tab. 5.1. Even though we did several adjustments to support gaming scene, till there is some issues that we haven't address in this thesis. For example, we now handles environmental luminance only and have not take complicated lighting in to consideration.

Compressor and decompressor. We implement different compression and decompression modules described in Chap. 4 in Matlab. We first parse the planar map into memory and leverage the pattern of barycentric coordinates. That is we use a code book to memorize those unclipped triangles, whose barycentric coordinates are always $(1, 0, 0, 1, 0, 0)$. Moreover, to optimize the compression ratio, we compress the face id and coordinates into different file because face id only evolve in integer where coordinates includes various data type according to compression modules.

Planar map streaming. Since packet loss in planar map streaming may cause a bunch of triangles damaged, we choose Transmission Control Protocol (TCP) instead of User Datagram Protocol (UDP) as transmission protocol. We then include (i) frame number, (ii) model view matrix, and (iii) draw call information into header and implement TCP socket using C++.

Control API. We implemented control API with TCP. We then define the transmission header with (i) user identification, (ii) time stamp, and (iii) command control, where

controls includes position movement and viewing direction change. The API implements with WIN socket and written in C++.

5.2 Setup

We compare our solution against the current cloud gaming platforms, in which all the rendering tasks are done at cloud servers. The rendered videos are compressed by the state-of-the-art video codec, such as x265 [43], and streamed to the clients. We do not compare against pure graphic streaming platforms, where all the rendering tasks are done at the clients. This is because for the (not-so-thin) clients that have enough horsepower to render game scenes, the benefits of cloud gaming are limited. For evaluating our distributed rendering platform, We compare our solution against the current cloud gaming platforms, in which all the rendering tasks are done at cloud servers. Fig. 5.1(a) indicates how the video-based solution works, where server transmit encoded video frame to clients. That is, all the rendering procedures take place in the server side, and a thin-client with a video decoders is sufficient. On the other hand, Fig. 5.1(b) indicates how graphic-based rendering platform works. In phase one, we first transmit 3D models, representing the scene, from server to client. And while playing (in phase two), vertex processor compute the vertex information according to viewing matrix, which is changing along with user commands. Finally, GPU renderer displays new frame to users using openGL pipeline. We do not compare against pure graphic streaming platforms, where all the rendering tasks are done at the clients. This is because for the (not-so-thin) clients that have enough horsepower to render game scenes, the benefits of cloud gaming are limited. We consider the following performance metrics in our experiment.

- **Gaming quality:** the rendered scene quality in PSNR and SSIM computed by comparing the original scene in games against the rendered one in different platform.
- **Bitrate:** the required transmission bandwidth for delivering data to clients across platforms.
- **Latency:** we measure latency in three different categories, network delay (ND), processing delay (PD), and playout delay (OD). Fig. 5.9 summaries the difference.

We consider diverse : (i) model complexity, basic or fine-grained bunnies, (ii) numbers of bunny, and (iii) moving speed, slow or fast (detailed in Tab. 5.1) scene to evaluate our proposed solution. In particular, we adopt 12 game scenes in our experiments, where our game scenes contain either basic bunnies, each with 37,677 vertices on average, or fine-grained bunnies, each with 200,700 vertices. We consider three performance metrics: video quality in PSNR and SSIM, bitrate in kbps, and component-wise running time in

ms. We run the experiments on an i7 3.4 GHz workstation with an NVidia Quadro M4000 card.

5.3 Results

Implications of diverse speed. We observe virtually no difference in terms of bitrate of slow and fast game scenes. Fig. 5.2 shows the bitrate consumptions under differnet bit-depth in different scenes. In particular, at a bit-depth of 8, the rate increase due to the speed is merely 0.04% on average. This can be attributed to the fact that we haven't leveraged the temporal redundancy, which is among our future tasks. In the following results, we show only results of slow scenes for making concrete observations.

Potential of our proposed solution. We first present the results from the bunnies at the low speed. We configure x265 [43] with ultra-fast preset and compress each scene with 6 different Quantization Parameter (QPs) to get its Rate-Distortion (R-D) curve. For our proposed solution, we exercise the tradeoff between bitrate and quality using the bit-depth of the uniform quantization. By varying the bit-depth between 5- and 9-bit, the resulting stream has different bitrates and video quality, which lead to the R-D curves. We first plot sample R-D curves from the scene in low spanning speed in Fig. 5.3. This figure shows that when bitrate is higher, the proposed solution outperforms x265 in terms of video quality in PSNR. We then apply another image quality metrics SSIM and get similar trend. Tab. 5.4 and Tab. 5.3 summarize SSIM scores in different scenes under the same bitrates.

Video quality improvement of our proposed solution. We next compare the video quality in PSNR of our solution against that of x265. Fig. 5.4 presents the bitrate of our solution under different bit-depths. Upon we derive the bitrate of each game scene, we derive the expected video quality of x265 by performing linear interpolation on its R-D curve. Fig. 5.5 plots the quality improvement of our proposed solution over x265. This figure shows that up to 5 dB improvement is possible, and as long as the bit-depth is ≥ 7 bits, our proposed solution results in higher PSNR.

Perceptual video quality metrics. Figs. 5.6 plot the video quality improvement of our proposed solution in SSIM with basic bunnies scenes and fine-grained bunnies scenes. The gap is as high as 0.14 in SSIM. For completeness, the raw video quality values are reported in Table 5.4, which supports the same finding. This may be attributed to two reasons. First, different from PSNR that only quantifies the *deviation* of signals, SSIM analysis the pictures structure and is for *perceptual* video quality. That is, our proposed solution results in higher perceptual video quality. Second, codecs, including H.265, are designed to maximize the PSNR value, rather than perceptual video quality.

Implications of complex game scenes. Our proposed solution may be more vulnerable to complex game scenes, compared to the existing cloud gaming platforms that stream videos at a fixed resolution. Our game scenes contain either basic bunnies, each with 37,677 vertices on average, or fine-grained bunnies, each with 200,700 vertices. Table 5.3 gives the raw video quality values from the scenes with fine-grained bunnies. Although our solution leads to poorer video quality than x265 in complex scenes, we argue that designers use simple scene with pre-rendered textures in practice gaming developing.

Per-component latency. We report the average delay per video frame in Table 5.2. We only consider the computationally intensive components, and with more complex game scenes (8 bunnies). This table shows that the latency of the client side component is much smaller compared to those of the server side components. We omit the network latency since the Internet conditions vary depending on sever/client location. Although the rendering time is measured on a workstation in our experiments, the negligible values of rendering component (≤ 0.83 ms on average) reveal that porting it to resource-constrained clients is possible.

Table 5.2: Running Time (ms), Average/Maximum, 8 Bunnies

	Server			Client
	Detection	Clipping	Occlusion	Rendering
Basic	0.27/0.28	25.56/33.41	1.94/2.68	0.22/0.48
F.G.	3.66/4.73	60.55/88.14	17.43/24.02	0.83/3.13

Implications of diverse speed and resolution. Distributed rendering supports homogeneous end devices naturally. We then compare the bitrate of our proposed solution and x265 at higher resolutions of 1080p, 2K, and 4K. Fig. 5.7 shows the R-D curve with four basic bunnies scene under different resolutions. We can observe that our proposed solution scale well to high resolution applications, such as 360 videos and VR. More precisely, fig. 5.8 reveals required bandwidth under a given PSNR scores. In fig. 5.8(a), our proposed solutions require less bandwidth in 4K. Moreover, our proposed method consumes less bandwidth scoring 30 PSNR, where PSNR 30 is concerned to be good image quality (see fig. 5.8(b)). This shows that our proposed solution has more potential in the future, where ultra-high-resolution displays become popular.

Table 5.3: Video Quality from Fine-grained Bunnies

Metric	**PSNR**		**SSIM**	
Scene	**Pro.**	**x265**	**Pro.**	**x265**
1 Bunny	32.4431	31.8702	0.9774	0.9081
4 Bunnies	32.3581	33.3911	0.9781	0.9666
8 Bunnies	32.8074	44.2013	0.9809	0.9999

Table 5.4: Video Quality from Basic Bunnies

Metric	PSNR		SSIM	
Scene	**Pro.**	**x265**	**Pro.**	**x265**
1 Bunny	32.37	29.41	0.98	0.84
4 Bunnies	31.66	28.49	0.85	0.04
8 Bunnies	31.21	27.94	0.97	0.87

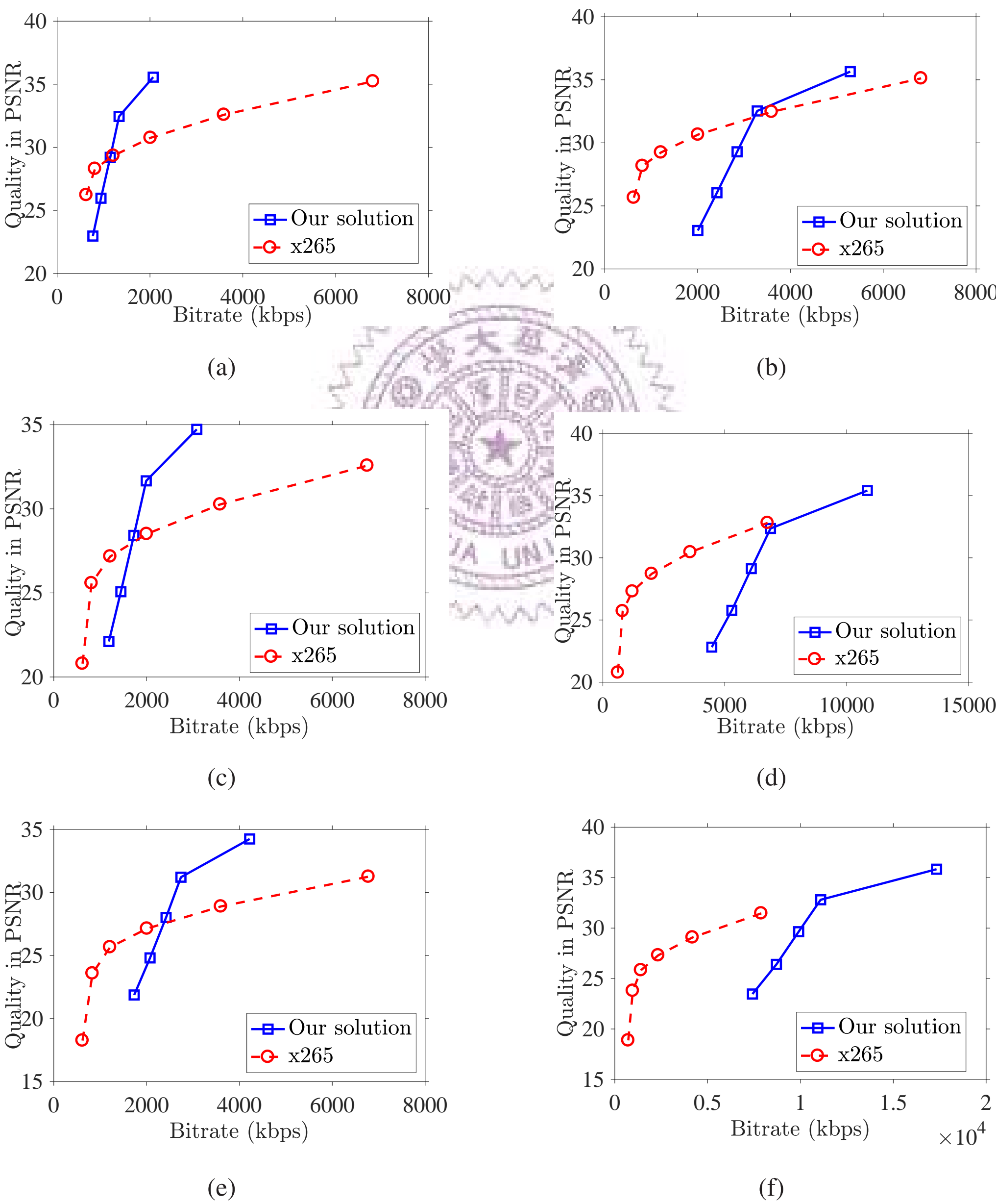
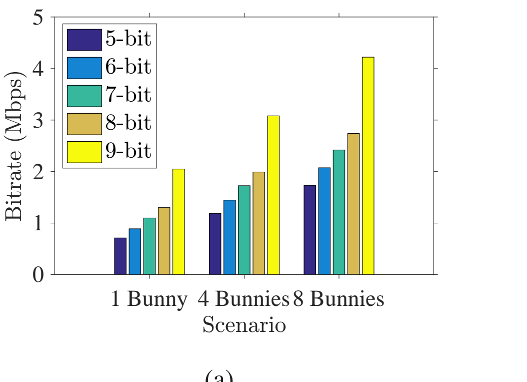
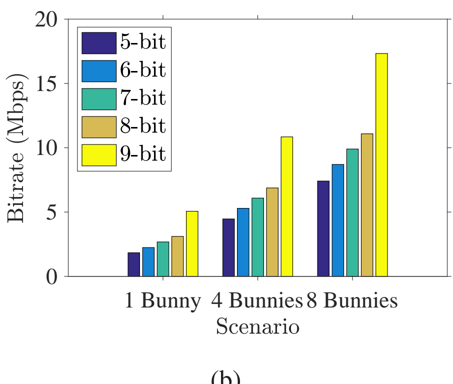


Figure 5.3: R-D curves of our proposed solution in PSNR compared with x265 in different scene: (a) one basic bunny, (b) one fine-grained bunny, (c) four basic bunny, (d) four fine-grained bunny, (e) eight basic bunny, and (f) eight fine-grained bunny.



(a)



(b)

Figure 5.4: Bitrate of our proposed solution at a different bit-depth of 8-bits,: (a)basic bunny scenes, and (b) fine-grained bunny scenes.

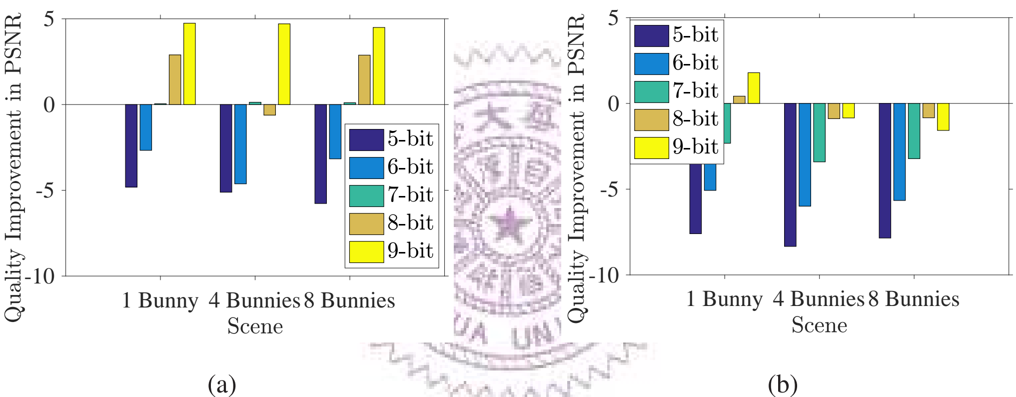


Figure 5.5: Performance improvement of our proposed solution in PSNR with (i) basic bunnies and (b) fine-grained bunnies.

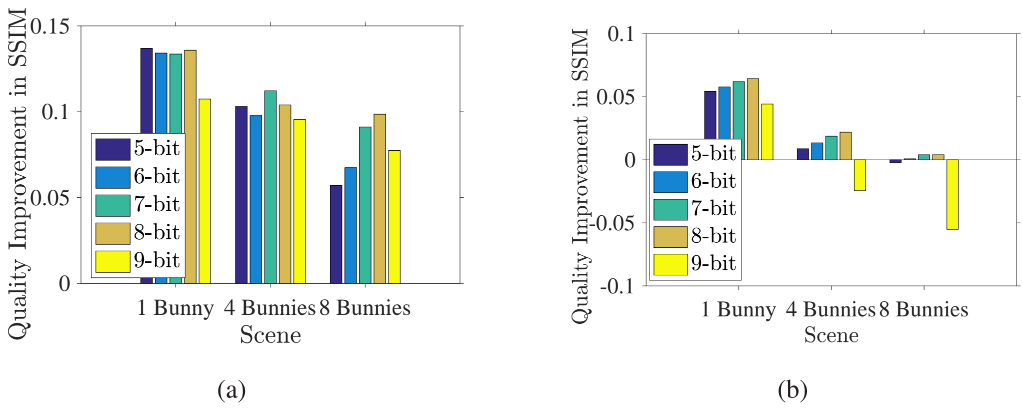


Figure 5.6: Performance improvement of our proposed solution in SSIM with (i) basic bunnies and (b) fine-grained bunnies.

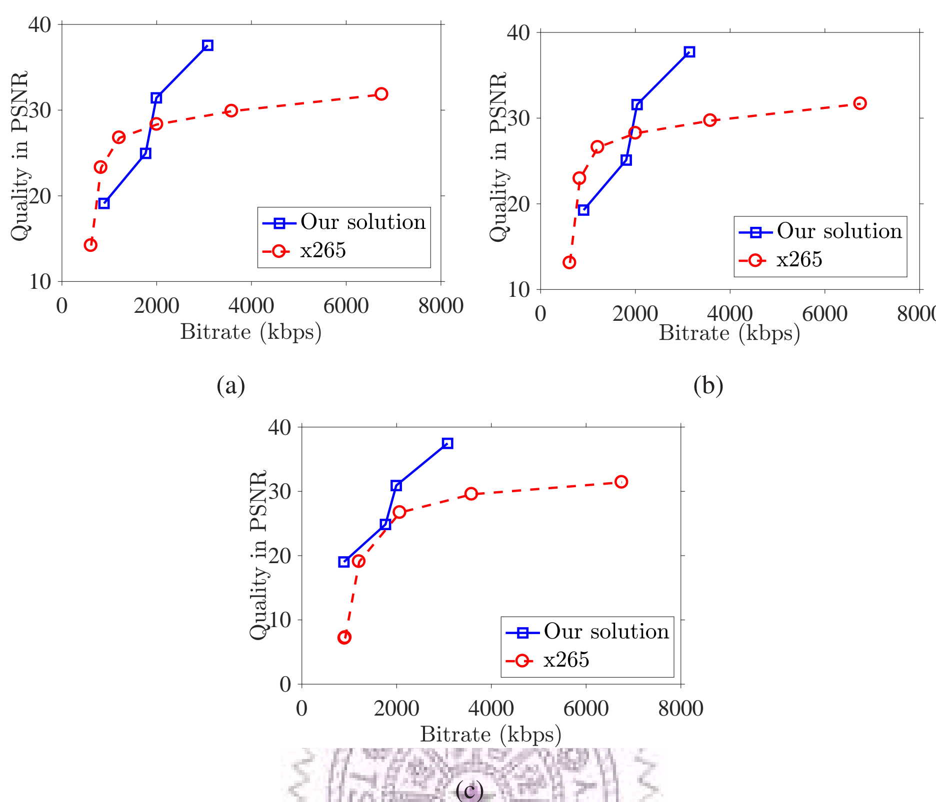


Figure 5.7: R-D curves of our proposed solution in PSNR compared with x265 in basic four bunnies scene in: (a) 1080p, (b) 2k, and (c) 4k resolution.

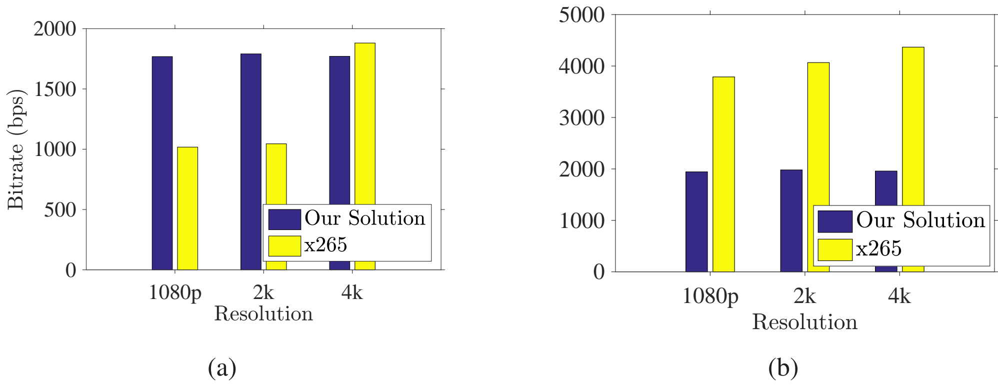


Figure 5.8: Bitrate required of our proposed solution and x265 with different resolution in: (i) 25 PSNR and (b) 30 PSNR.

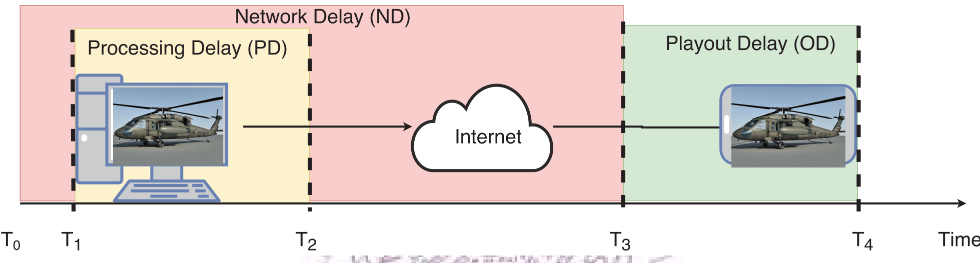


Figure 5.9: Delay categories and its definition. T_0 represents the time client sends the command to sever, where T_1 is the time server receives. Meanwhile, sever transmit encoded planar map at T_2 and client obtained the data at T_3. Finally, the reconstructed scene is played at T_4.

Chapter 6

Related Work

In this chapter, we are going to survey related literature under three direction: (i) clouding gaming platform, and (ii) mesh compression.

6.1 Cloud Gaming Platform

Cloud gaming platforms [17, 19, 21] can be roughly divided into three groups [6]: (i) video, (ii) graphics, and (iii) hybrid streaming. Video streaming refers to gaming platforms in which each frame is rendered completely on the server, and the frames are compressed into video streams for transmission to the client. The problem with this approach stems from the tradeoff made between bitrate and image quality. Graphics streaming describes transmission of game scene data and/or rendering commands of each frame from server to client. This approach can produce the highest rendering quality, but the graphics hardware available on the client must be capable of high performance rendering, which is not always the case, especially in cloud gaming systems. Hybrid streaming as implied by the name refers to some clever combination of the above technologies. In this thesis, we explore a new graphics streaming approach using planar maps for cloud gaming.

Hemmati et al. [17] argue that client-side rendering requires more transmission bandwidth and computational power than server-side rendering. Moreover, server-side rendering is more adaptive to various network conditions. Therefore, they adopt the video streaming approach, and propose to reduce the bandwidth consumption by prioritizing objects in 3D scenes and assigning bitrates to different objects accordingly.

Jurgelionis et al. [21] propose and evaluate a graphics transmission approach, which renders at the client side if a GPU is available, and falls back to video streaming otherwise. Their design decision is based on the observation that cloud gaming is interaction-intensive and delay-intolerant. Thus, even though client side rendering provides the highest graphics fidelity, providing a fallback system is necessary.

There are several compression algorithms proposed for graphics streaming. For example, Meilander et al. [25] propose: (i) a caching mechanism for rendering commands, (ii) a compression algorithm for rendering instructions, and (iii) multi-layer representation of 3D objects. Lin et al. [23] utilize both intra- and inter-frame compression of 3D models to reduce the required transmission bandwidth. Nan et al. [29] introduce a hybrid delivery approach, where the server progressively streams the encoded frames and the graphics information. Similarly, Chuah et al. [9] aim to fully leverage the computational power on the client by rendering the low-quality base layer locally, while the server transmits a high-quality enhancement layer.

The above-mentioned cloud gaming studies [9, 17, 21, 23, 25, 29] are not exhaustive, interested readers are referred to a survey paper [6]. Compared to the existing cloud gaming platforms, our platform: (i) naturally achieves distributed rendering and brings cloud gaming to inexpensive computing devices with weak GPUs, (ii) produces and compresses concise data representations for clients with limited network bandwidth, and (iii) scales to high-resolution game scenes for large displays.

6.2 Mesh Compression

With the emerging applications of 3D transmission, such as VR, the technologies of compressing 3D mesh becomes more and more important. Peng et al. [34] surveys the start-of-art 3D compression algorithm. To start with, mesh is a sequence of triangles, whereas not only vertex information but also connectivity relationship are included. Since this thesis compress data type, which is do not include the connectivity information, literature regarding to connectivity compression are listed for reference [13, 16, 41, 42]. Geometry coding can further be classified into two groups: single-rate, and progressive.

Different from connectivity compression, geometry compression could be lossy. To exploit the fact that adjacent vertices are highly correlated, most compressions follow a procedure: quantization on vertex, prediction of quantized data, and lossless encoding. Quantization method includes uniform/non-uniform and non-uniform includes vector and scale quantization [15]. In single-rate geometry compression, most used quantization module is scale quantization. In [13,33,41] encode meshes in 8 to 16 bit quantization resolution globally. Chow. [8] propose a quantization resolution that is adaptively changed according to the input blocks of triangles. Secondly, prediction modules is then applied to further reduce required bitrate by leveraging the spatial property. Common used prediction includes delta prediction, linear prediction, parallelogram prediction and second-order prediction. Finally, lossless coding techniques, such as Huffman encoding and other entropy coding in. [11]

Progressive meshes, proposed by Hoppe. [18], supports scheme that transmit a basic mesh first and then gradually improve its quality by transmit more meshes. When it comes to compression, it removes vertices and triangles iteratively, and in returns, add back iteratively when transmission. Li and Kuo [22] first introduce a coder that remove a vertex at a time while Cohen et al. [10] further improves it by removing a set of vertices. Advance data structure is then applied to progressive compression. Peng and Kuo. [35] use octree structure to encode a mesh. That is each level of tree represents different level of sophistication of mesh.

In this thesis, we study those compression modules above and decide the module parameters through real experiments on a brand new data type.

Chapter 7

Discussion

7.1 Inter frame compression

In this thesis, we concentrate on intra-frame planar map compression and leave inter-frame planar map as a future work. While surveying the literature on inter-frame mesh compressions, we find few studies compare to intra-frame compression. Work from Yamasaki and Aizawa [44] named inter-frame mesh compress as time-varying mesh and divide those meshes into patches to leverage the redundant information across frames. Due to 3D models complicated correlations, they develop a patch matching algorithm and use vector quantization plus entropy coding in compression pipeline.

However, time-varying mesh compression algorithm may not suitable in planar map with the following two reason: (i) the patch matching algorithm is designed to 3D meshes and cause unnecessary computation overhead on 2D planar map, (ii) planar map is represented in barycentric coordinates and do not share the spatial properties for compression. We then take a closer look to time varying planar map and find that planar map includes both clipped and unclipped triangle and in terms of the world coordinate, those unclipped triangles are most likely to remain unclipped in near-by frames. Therefore, we may use the properties of unclipped triangles to develop more sophisticated planar map compression.

7.2 Integration with Game Engine

As our goal is to implement a real cloud gaming platform, integrating with game engine is an important issue worth discussion. Messaoudi et al. [27] dissecting and analysis one of the most used gaming engine - Unity3D. The authors divide game engine into the following modules: (i) artificial intelligence, (ii) physics engine, (iii) scripting, (iv) input, (v) multimedia rendering, and (vi) networking. In case of integrating with a game engine,

our proposed pipeline serves as a combination of multimedia rendering and networking modules. That is, Unity3D or other game engine builds the graphical element as well as other game logics, whereas our pipeline renders whatever in the scene given by the game engine. To be more specific, our pipeline requires the following inputs from game engine: (i) 3D objects, (ii) gamers' viewing information, and (iii) lighting sources. With those inputs we can then rendering and interact with gamers using our proposed pipeline.

Therefore, our next step is to integrate with existing open source game engine, such as ORGE [32] and mini3D [31]. Before integrating with game engine, we need to slightly revise our propose vertex rendering modules to support complicated light source and to receive dynamic lighting configuration from game engine. After integrating with game engine, we can then evaluate end-to-end performance and further recognize research problems in the future.

7.3 Holes Filling Issue and User Study

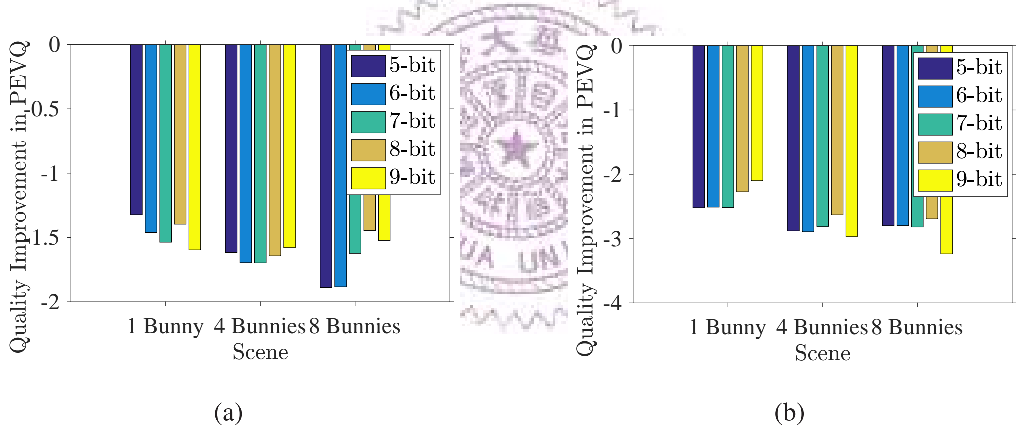
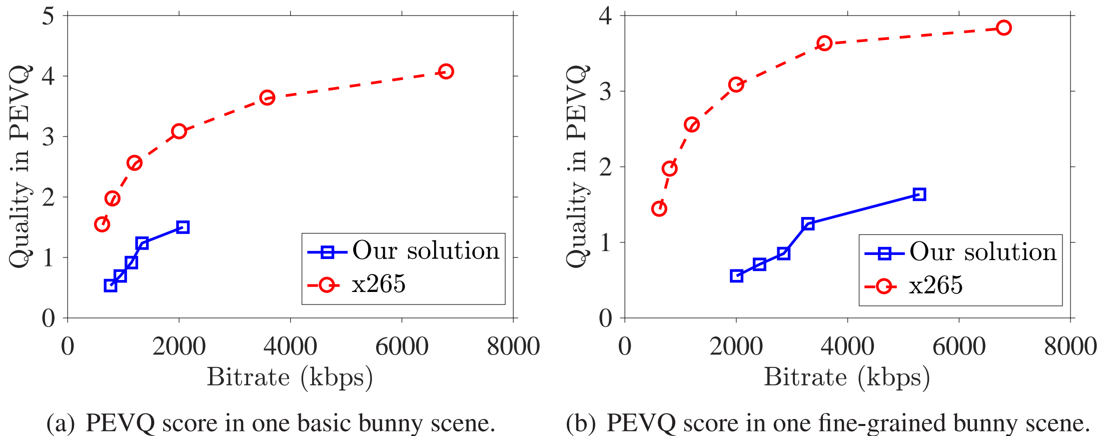


Figure 7.1: Performance improvement of our proposed solution in PEVQ with (i) basic bunnies and (b) fine-grained bunnies.

In Chap. 5, we evaluate our proposed method with common used video quality metrics, such as PSNR and SSIM, where PSNR evaluate the image degradation pixel-wised and SSIM analysis the structures of image. Yet, in a interactive application, we aim to predict user's viewing experience before putting into market. We then evaluate the rendered scene quality in OPVQ, computed by comparing the original scene in games against the rendered one in different platform. OPVQ is an algorithm of Perceptual Evaluation of Video Quality (PEVQ) described in ITU-T J.247 Annex B [20]. Here, we use Skarseth et al's. [40] open source package as OPVQ toolkit.

We found that our proposed method performs poorly compared with x.265 under the same bandwidth constraints. We then take a closer on the evaluation results by plotting the one basic bunny scene and the one high quality bunny scene in Fig. 7.2. We than

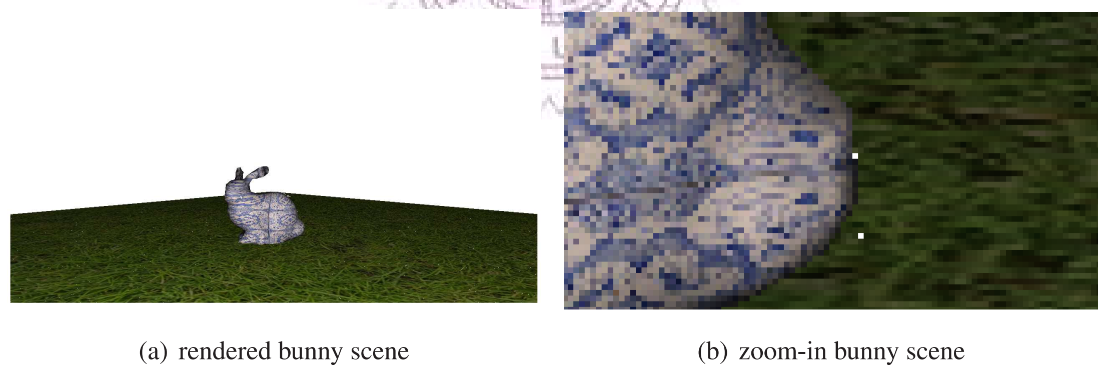
found that even giving a lot more bandwidth, the improved PEVQ scores is very limited and considered it as a natural limitation from our proposed method.



(a) PEVQ score in one basic bunny scene. (b) PEVQ score in one fine-grained bunny scene.

Figure 7.2: One bunny rendered scene in PEVQ scores.

Fig. 7.3 is a rendered scene from the basic one bunny scene. When zooming in the rendered scene, we find there is small holes. There are two reasons generating small holes in our proposed method: (i) when generating 2D planar map, there occasionally facing some occlusion on clipping modules, and (ii) when compress and decompress the planar map, there is some loose compression and cause small misplacement on vertex.



(a) rendered bunny scene (b) zoom-in bunny scene

Figure 7.3: One bunny rendered scene.

Chapter 8

Conclusion

We studied the feasibility of leveraging the 2D planar map streaming and distributed rendering in cloud gaming. We first presented the server and client pipelines, based on the standalone 2D planar map rendering pipeline with additions of several components for compression and transmission. We then dived into the core challenge of the platform: the design of the compressor/decompresser of 2D planar maps, which has not been studied before. We designed a parameterized compression component, and derived the optimal parameters through real experiments. We then put up the rendering pipelines in our platform, and compared its performance against the state-of-the-art x265. Our evaluation results are quite promising. Although our platform is outperformed by x265 in PSNR at low bitrate, we significantly outperform it at high bitrates. In addition, our platform outperforms x265 in terms of video quality, e.g., by up to 0.14 in SSIM. Other merits of the proposed platform include: (i) fast running time, especially at the client side and (ii) high scalability to ultra-high resolutions without bitrate penalty.

The current work can be extended in several directions. First, to address the problem of low PEVQ scores, we may use morphological antialiasing [38] to fill the holes in rendered scenes, based on the precise silhouette information from planar map generation. Second, the platform can be extended to support foveated rendering, which renders a part of the region at full quality, and others at degraded quality. Last, we can analyze gamers' mobility pattern to leverage temporal redundancy in our planar map streams. These enhancements will further improve the performance of our platform.

Bibliography

[1] D. Arthur and S.Vassilvitskii. k-means++: The advantages of careful seeding. In *Proc. of the ACM-SIAM Symposium on Discrete algorithms (SODA'07)*, pages 1027–1035, 2007.

[2] P. Asente, M. Schuster, and T. Pettit. Dynamic planar map illustration. *ACM Transactions on Graphics*, 26(3):30, 2007.

[3] P. Baudelaire and M. Gangnet. Planar maps: an interaction paradigm for graphic design. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems (CHI'89)*, pages 313–318, 1989.

[4] G. Bernstein and D. Fussell. Fast, exact, linear booleans. *Computer Graphics Journal*, 28(5):1269–1278, 2009.

[5] W. Cai, R. Shea, C. Huang, K. Chen, J. Liu, V. Leung, and C. Hsu. The future of cloud gaming. *Proceedings of the IEEE*, 104(4):687–691, 2016.

[6] W. Cai, R. Shea, C. Huang, K. Chen, J. Liu, V. Leung, and C. Hsu. A survey on cloud gaming: future of computer games. *IEEE Access*, 4:7605–7620, 2016.

[7] W. Cheng and 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 (NOSSDAV'08)*, pages 9–14, 2008.

[8] M. Chow. Optimized geometry compression for real-time rendering. In *Proc of Visualization*, pages 347–354. IEEE, 1997.

[9] S. Chuah, N. Cheung, and C. Yuen. Layered coding for mobile cloud gaming using scalable blinn-phong lighting. *IEEE Transactions on Image Processing*, 25(7):3112–3125, 2016.

[10] D. Cohen, D. Levin, and O. Remez. Progressive compression of arbitrary triangular meshes. In *Proc. of Visualization*, volume 99, pages 67–72, 1999.

[11] T. Cover and J. Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[12] M. Deering. Geometry compression. In *Proc. of Conference on Computer Graphics and Tnteractive Techniques (SIGGRAPH'95)*, pages 13–20, 1995.

[13] M. Deering. Geometry compression. In *Proc. of Computer graphics and interactive techniques*, pages 13–20. ACM, 1995.

[14] A. Ellis, W. Hunt, and J. Hart. Svgpu: real time 3d rendering to vector graphics formats. In *Proc. of High Performance Graphics (HPG'16)*, pages 13–21, 2016.

[15] A. Gersho and R. Gray. Vector quantization I: Structure and performance. In *Vector quantization and signal compression*, pages 309–343. Springer, 1992.

[16] S. Gumhold and W. Straßer. Real time compression of triangle mesh connectivity. In *Proc. of Computer graphics and interactive techniques*, pages 133–140. ACM, 1998.

[17] M. Hemmati, A. Javadtalab, A. Shirehjini, S. Shirmohammadi, and T. Arici. Game as video: bit rate reduction through adaptive object encoding. In *Proc. of ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'13)*, pages 7–12, 2013.

[18] H. Hoppe. Progressive meshes. In *Proc. of the Computer graphics and interactive techniques*, pages 99–108. ACM, 1996.

[19] C. Huang, K. Chen, D. Chen, H. Hsu, and C. Hsu. GamingAnywhere: the first open source cloud gaming system. *ACM Transactions on Multimedia Computing, Communications, and Applications*, 10(1s):10:1–10:25, January 2014.

[20] Objective perceptual multimedia video quality measurement in the presence of a full reference. Standard, ITU Telecommunication Standardization Sector, 2008.

[21] A. Jurgelionis, P. Fechteler, P. Eisert, F. Bellotti, H. David, J. Laulajainen, R. Carmichael, V. Poulopoulos, A. L. P. Perälä, A. Glora, and C. Bouras. Platform for distributed 3d gaming. *International Journal of Computer Games Technology*, 2009:1–15, 2009.

[22] J. Li and C. Kuo. Progressive compression of 3D graphic models. In *Proc. of Multimedia Computing and Systems*, pages 135–142. IEEE, 1997.

[23] L. Lin, X. Liao, G. Tan, H. Jin, X. Yang, W. Zhang, and B. Li. Liverender: a cloud gaming system based on compressed graphics streaming. In *Proc. of ACM International Conference on Multimedia (MM'14)*, pages 347–356, 2014.

[24] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[25] D. Meiländer, F. Glinka, S. Gorlatch, L. Lin, W. Zhang, and X. Liao. Bringing mobile online games to clouds. In *Proc. of IEEE Computer Communications Workshops (INFOCOM WKSHPS'14)*, pages 340–345, 2014.

[26] R. Mekuria, M. Sanna, S. Asioli, E. Izquierdo, D. Bulterman, and P. Cesar. A 3d tele-immersion system based on live captured mesh geometry. In *Proc. of the ACM Multimedia Systems Conference (MMSys'13)*, pages 24–35, 2013.

[27] F. Messaoudi, G. Simon, and A. Ksentini. Dissecting games engines: the case of Unity3D. In *Network and Systems Support for Games (NetGames), 2015 International Workshop on*, pages 1–6. IEEE, 2015.

[28] D. Mishra, M. Zarki, A. Erbad, C. Hsu, and N. Venkatasubramanian. Clouds+ games: A multifaceted approach. *IEEE Internet Computing*, 18(3):20–27, 2014.

[29] X. Nan, X. Guo, Y. Lu, Y. He, L. Guan, S. Li, and B. Guo. A novel cloud gaming framework using joint video and graphics streaming. In *Proc. of IEEE International Conference on Multimedia and Expo (ICME'14)*, pages 1–6, 2014.

[30] A. Nordland. Compression of 3D media for internet transmission. Master's thesis, University of Oslo, 2016.

[31] May 2017. `http://mini3d,org`.

[32] May 2017. `http://orge3d,org`.

[33] C. B. V. Pascucci and G. Zhuang. Single-resolution compression of arbitrary triangular meshes with properties. In *Proc. of Data Compression Conference (DCC'99)*, pages 247–256. IEEE, 1999.

[34] J. Peng, C. Kim, and C. Kuo. Technologies for 3D mesh compression: A survey. *Journal of Visual Communication and Image Representation*, 16(6):688–733, 2005.

[35] J. Peng and C. Kuo. Geometry-guided progressive lossless 3D mesh coding with octree (OT) decomposition. 24(3):609–616, 2005.

[36] PlayStation Now web page, January 2015. `http://www.playstation.com/en-us/explore/playstationnow/`.

[37] January 2017. `https://www.playstation.com/en-gb/explore/playstation-now/faq/`.

[38] A. Reshetov. Morphological antialiasing. In *Proc. of Conference on High Performance Graphics (HPG'09)*, pages 109–116, 2009.

[39] P. Ross. Cloud computing's killer app: Gaming. *IEEE Spectrum*, 46(3):14–14, 2009.

[40] K. Skarseth, H. Bjørlo, P. Halvorsen, M. Riegler, and C.Griwodz. OpenVQ: a video quality assessment toolkit. In *Proc. of ACM International Conference on Multimedia (MM'16), OSSC paper*, pages 1197–1200, 2016.

[41] G. Taubin and J. Rossignac. Geometric compression through topological surgery. *ACM Transactions on Graphics (TOG)*, 17(2):84–115, 1998.

[42] C. Touma and C. Gotsman. Triangle mesh compression. *Proc. of graphics interface*, pages 26–34, 1998.

[43] February 2017. `http://x265.org`.

[44] T. Yamasaki and K. Aizawa. Patch-based compression for time-varying meshes. In *Proc. of IEEE International Conference on Image Processing (ICIP'10)*, pages 3433–3436, 2010.

TSING HUA UNIVERSITY