# Hybrid Cloud Rendering System for Massive CAD Models

André Moreira, Paulo Ivson, Waldemar Celes
Tecgraf Institute and Computer Science Department
Pontifical Catholic University of Rio de Janeiro - PUC-Rio
Rio de Janeiro, Brazil
Email: {asouza, psantos, celes}@tecgraf.puc-rio.br

***Abstract*—The recent advances in cloud services enable an increasing number of applications to offload their intensive tasks to remote computers. Cloud rendering comprises a set of services capable of rendering a 3D scene on a remote workstation. Notable progress in this field has been made by cloud gaming services. However, a gap remains between existing cloud rendering systems and other graphics-intensive applications, such as visualization of Computer-Aided Design (CAD) models. Existing cloud gaming services are not suitable to efficiently render these particular 3D scenes. CAD models contain many more objects than a regular game scene, requiring specific assumptions and optimizations to deliver an interactive user experience. In this work, we discuss and propose a novel hybrid cloud rendering system for massive 3D CAD models of industrial plants. The obtained results show that our technique can achieve high frame rates with satisfactory image quality even in a constrained environment, such as a high latency network or obsolete computer hardware.**

## I. Introduction

Cloud rendering, also known as remote rendering, comprises a set of services for graphical content generation provided by remote computers. Once this service is requested, the resulting image or video is transmitted to the client and finally displayed. Some remote rendering systems also send others types of data, for instance, proxy geometries to be rendered on the client side [1]. Recent advances in both hardware capabilities and cloud solutions are continually pushing the boundaries of cloud rendering systems. Cloud gaming [2], for example, has gained much attention in the last years due to growing market demand. These services allow the latest games to be played without the need for last generation hardware.

On the other hand, when looking beyond the game domain, there is a lack of studies about how to employ cloud rendering services effectively for other visualization applications. For example, the Architecture, Engineering & Construction (AEC) industry mainly uses Computer-Aided Design (CAD) in different phases of the life-cycle of engineering projects. These models play several roles, such as virtual construction planning, in order to reduce costs and shorten deadlines. As a result, they must contain a high level of detail to represent the real world accurately. This amount of fine-grained geometries demand cutting-edge computers for efficient rendering.

In order to render massive 3D models in thin clients (low-cost devices), it is necessary to offload this burden to a powerful remote computer. Cloud rendering services can be employed to mitigate this limitation. Besides, employing cloud rendering on CAD models brings the following benefits:

- The models are stored in a central repository on the server. Hence the users do not need to gain direct access to these files, reducing the cases of confidential data misappropriation. In addition, this centralization reduces problems related to model versioning and updating. This is a fundamental benefit since it is very common the occurrence of inconsistent information from different teams in large-scale engineering projects.
- Cloud rendering makes use of mobile devices feasible. In addition, power consumption is lower than it would be if the whole rendering was done on the client side.
- Since a cloud rendering engine runs on a well-known device, the engine can provide state-of-the-art rendering algorithms and optimizations to the users.

Existing cloud rendering systems are not ready to support 3D CAD models since these services lack the ability to handle large-scale interactive scenes. In this paper, we propose a novel hybrid cloud rendering service that fills this gap. The hybrid approach consists in performing most of graphics processing on the server side, while the client addresses lightweight rendering commands, ensuring an interactive application even on a constrained network. Our new technique paves the way for many research opportunities related to predictive rendering and automated load balancing that could further improve Quality of Experience (QoE) of cloud rendering systems in general.

### A. Contributions

Our hybrid approach brings the following advantages:

1) *Final image with better quality*: the image produced on the server is compressed before transmission. This process reduces the image quality. On the other hand, the image produced by the client is not compressed, preserving its high fidelity. Hence, the combination of these two images results in a more pleasant looking image than if it was fully rendered on the server side.
2) *Lightweight geometry representation*: instead of using the traditional triangular meshes for all scene representation, we represent CAD objects by their implicit forms as much as possible. This representation is much

more compact than triangular meshes, enabling efficient transmission and storage, especially on the client side.

3) *Less dependency on network condition*: since some parts of the 3D model are stored and rendered locally, the proposed framework maintains application responsiveness even when facing a constrained network scenario. This responsiveness is achieved by displaying the partial image produced on the client side as soon as it is ready. In this case, the user can still orient himself, making it possible to continue navigating through the scene.
4) *Greater scalability*: although one of the major contributions of cloud rendering systems is enabling thin devices to handle massive models, high-end computers can also take advantage of our solution. In this case, the hybrid approach can fully take advantage of the available processing power by assigning more jobs to be performed on the client side. Consequently, the burden on the server is reduced, allowing it to handle larger scenes and/or more users.

The rest of the paper is organized as follows. We discuss related works in Section II. In Section III we present an overview about Cloud Rendering Systems and talk specially about their latency and robustness analysis. In Section IV-A, we discuss efficiency in CAD Model rendering. Our proposed method is presented in details in Section IV and our results are evaluated in Section V. Finally, in Section VI we summarize our method and point out further improvements.

## II. Related Work

Cloud rendering systems have been widely used in different fields, ranging from scientific visualization to game industry. In this section, we discuss some of the most notable works.

The earliest remote rendering systems focused on 2D graphics, for example the X Window System [3]. Research focused mainly on reducing the refresh rate of the windowing system [4]–[7]. More recently, distributed rendering systems have emerged [8]–[10]. In these approaches, the screen is divided into tiles, which are dispatched to a remote computer to be drawn. Once all tiles have been processed, the resulting images are stitched together and the final result is displayed to the user. It is important to note that these works are only concerned about how to divide, distribute and process the tasks effectively.

The recent proliferation of high-end cloud services, as well as the massive adoption of high-bandwidth network, enabled the creation of powerful 3D remote rendering systems. WireGL [11] is a scalable graphics system that allows users to send graphics commands to graphics servers. Inspired by the WireGL system, the Chromium project [12] have emerged providing an interface to control the graphics commands on clusters. This framework lead to a mobile-devices-oriented solution which was used on server side to manage the rendering commands [13].

Remote rendering has also been employed for scientific data (volume rendering) [14]. The authors made use of a hybrid rendering approach where the schedule was determined by data processing and transfer times. The proposed solution kept several versions of the same volume data, varying the level-of-detail (LOD) of each. When the network condition changed, the server sent gross-resolution models, and more refined versions were delivered progressively to the client when the network improved. The drawback of this approach is the constant transmission of the 3D models, congesting the network.

Cloud gaming is a field that is recently experiencing a significant expansion. In this category, we can list OnLive [15] and GamingAnywhere [16] services. The GamingAnywhere is an open-source cloud gaming service that adopts the H.264 standard for encoding and decoding the transferred images. The server is responsible for all graphics processing, including the game execution. The rendered images are encoded by the server and then streamed to the client. The client simply decodes and displays the received images.

The aforementioned solutions are not well suited for 3D CAD scenes since they are mostly general-purpose services, lacking specific algorithms and optimizations to handle massive models made of many fine-grained geometries. Since our solution targets a specific domain, we can take advantage of additional rendering optimizations.

## III. Cloud Rendering Systems

### A. Overview

Considering a generic cloud rendering system, the client is responsible for listening to and interpreting user inputs. Once a user interaction happens, the client processes this interaction and requests a new frame to the server. The server addresses the requisition, updates its internal state and finally produces an image to send back. The client receives this image, decodes it and displays the final result onto the screen. Depending on the type of data that is transmitted from the server to the client, the system can be classified into two major categories: model-based or image-based.

Model-based systems send geometric information to the client, typically as triangular meshes. The size of the transmitted data is proportional to how complex or detailed the underlying representations are. In contrast, image-based systems render the scene entirely on the server side and only send resulting images to the client. Unlike model-based systems, the size of transmitted data mostly depends on screen resolution, regardless of the complexity of the scene. This property made image-based systems popular, since they are more robust to the variation on the network quality. A typical image-based rendering workflow is depicted in Fig. 1.

Both image-based and model-based approaches comprise many other different techniques. Among image-based techniques, the Depth-Image-Based Rendering (DIBR) [17] is the most commonly implemented on modern cloud rendering systems, since it can produce satisfactory results with low processing cost. Basically, the server generates and transmits both depth and color images. These two images can be used by the client to promptly generate new images from other viewpoints. However, the drawback of this approach is the

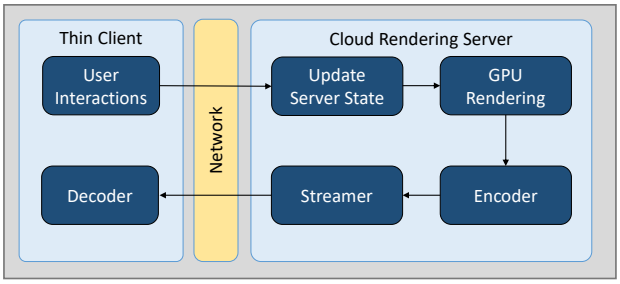



Fig. 1. The phases involved in a typical cloud rendering system

presence of holes in the final image. These holes are inevitable and they arise from occluded surfaces on the reference image or when the reference image is undersampled. Nonetheless, DIBR techniques are very useful to significantly reduce the latency perceived by the user.

### B. Latency Analysis

The ultimate challenge of any cloud rendering system consists in reducing *Response Delay* ($RD$). It is defined as the elapsed time from the user interaction to the display of the resulting image on the screen. The cloud system must ensure low latency rates, preventing the degradation of the Quality of Experience (QoE) [18]. According to [19], the tolerable delay for first-person shooter games is around 100ms, or approximately ten frames per second (*fps*). The reduction of system latency is achieved by shortening the delay of each step involved in the production of the final image. When considering the system expressed in Fig. 1, the *RD* is defined as the sum of three components:

$$RD = ND + SD + CD, \tag{1}$$

where:

- **Network Delay** (*ND*): time required by the client to transmit user events and by the server to send the produced image. It is also known as network round-trip time.
- **Server Delay** (*SD*): time from when a user event arrives at the server to when its corresponding image is produced.
- **Client Delay** (*CD*): difference between the time when the client receives a server response and the time the remote image is displayed to the user.

Cloud rendering systems are very susceptible to the network performance since the response delay depends on the network delay. In these situations, a hybrid approach enables the client to display intermediary results, regardless of whether the remote frame is available. A small part of the scene is rendered locally, while the remainder is computed on the remote server. While the server is busy, the client renders its partial image and combines it with the DIBR of the last available remote frame. As soon as the client receives the remote image, it updates the final results to the user. This way, the client need not wait for the server's response to provide a feedback to the user's input. Clearly, this approach can be very useful when the system faces a network with limited bandwidth.

TABLE I
DESCRIPTIONS OF THE NOTATIONS USED FOR LATENCY ANALYSIS

| Symbol | Description |
|---|---|
| $v$ | The viewpoint. $v_s$ and $v_c$ are the server and client viewpoint, respectively. |
| $R^{v_i}$ | The rendered image for a given viewpoint $v_i$. |
| $rdr(v_i)$ | The rendering time to draw a frame from viewpoint $v_i$. $rdr_s$ and $rdr_c$ are the server rendering time and client rendering time, respectively. |
| $enc(R^{v_i})$ | The elapsed time for encoding an image $R^{v_i}$. |
| $dec(R^{v_i})$ | The elapsed time for decoding an image $R^{v_i}$. |
| $W^{v_s \to v_c}$ | DIBR on the viewpoint $c$ using a reference image from viewpoint $s$. |
| $cmb$ | The time to produce the final image combining the local and remote images. |

Because of this behavior, the response delay with hybrid rendering only depends on the processing time on the client side. The *SD* and *ND* only affect the image quality. Therefore, considering the notation defined in Table I, the response delay (*RD*) and the remote frame delay (*RF*) can be expressed as:

$$RF = rdr_s(v_s) + enc(R_s^{v_s}) + dec(R_s^{v_s}) + ND \tag{2}$$

$$RD = CD = rdr_c(v_c) + W^{v_s \to v_c} + cmb(R_c^{v_c}, R_s^{v_c}) \tag{3}$$

### C. Requirements for Robust Cloud Rendering Systems

In order to achieve a robust cloud rendering system, we established the following requirements:

1) **Lightweight data transmission:** The data exchanged over the network must be as compact as possible, avoiding network congestion and preventing the client from being idle while waiting for the necessary data.
2) **Efficient rendering:** massive 3D CAD models demand large graphics memory (VRAM) and high processing power. Despite being a critical step, the rendering is one of the most controllable steps. Therefore, it is important to ensure an efficient rendering mechanism, such as reducing synchronization points between CPU and GPU. These synchronizations are usually the bottleneck in rendering applications.
3) **Controlled workload on the client side:** The client's workload must be proportional to its processing power. The server's capabilities are useless if the client is constantly busy processing its tasks and can barely consume the remote data.
4) **Low response delay, even on a constrained network:** The rendering system must not rely upon network conditions in order to provide some feedback to the user. Otherwise, the system can suffer with high latency changes. Partial scene rendering results are allowed, as they convey a spatial notion to the user.

In the next sections, we present in detail our proposal for a cloud rendering system for CAD models, which was designed taking into account each one of these requirements.

TABLE II
LIST OF SUPPORTED IMPLICIT GEOMETRIES

| Geometry Type | Geometry Description | Total Size (bytes)[1] |
|---|---|---|
| • Parallelograms<br>• Spheroids<br>• Ellipsoids | **M** | 36 |
| • Cylinders | **M** + 2 offsets + radius + height | 52 |
| • Sloped Cylinders | **M** + radius + height + 4 slopes | 60 |
| • Truncated Cone | **M** + 2 offsets + 2 radii | 52 |
| • Square Frustum | **M** + 2 offsets + 4 side lengths | 60 |
| • Rectangular Torus<br>• Circular Torus | **M** + 2 radii + sweep angle | 48 |

## IV. PROPOSED HYBRID CLOUD RENDERING SYSTEM

Our goal was to design a system that matches all requirements established in Subsection III-C. We discuss the details of our methodology from four different points of view: the CAD rendering algorithm, the communication model, the client architecture and the server architecture.

### A. *Efficient CAD Model Rendering*

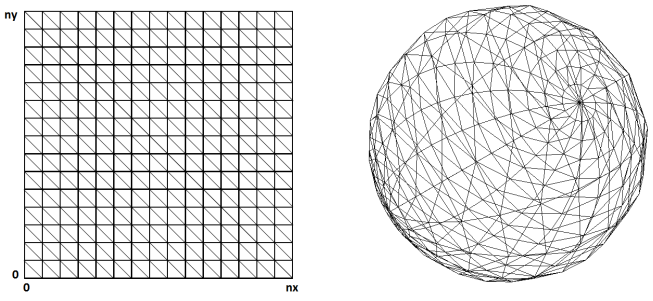
As stated before, CAD models provide a unique challenge for interactive cloud rendering systems due to their massive number of objects. A striking feature is their repetition, that is, many objects are instanced multiple times. Typical CAD scenes can be mostly described by a collection of simple solids: spheres, parallelograms, cones and others. Moreover, these solids can be stored in memory using their implicit representation instead of their triangular meshes. These peculiarities play a central role in our cloud rendering methodology. The implicit representation is much more lightweight when compared to triangular meshes.

If we consider a non-indexed triangular mesh, each triangle is composed by three vertices. Each vertex has two attributes: position and normal, each one represented by three floating point scalars. A mesh with $k$ triangles would occupy $k \times 72$ bytes[1]. Moreover, at least 9 additional floats (36 bytes) may be necessary to represent the mesh transformations: translation, rotation, scale (36 bytes). For the sake of simplicity, we denote these transformations as **M**. In contrast, those same 36 bytes (**M** values) can store the entire implicit representation of an ellipsoid, regardless of its resolution.

Our system supports nine different implicit representations. Some of them are defined only by their **M** transformations, such as the ellipsoid, but other geometries need some additional parameters. All implicit geometries supported by our system and their respective attributes are listed in Table II.

The developed hybrid cloud rendering system uses these compact representations to meet the requirement of a lightweight transmission mechanism. This format remains compatible with outdated GPUs, since the system only needs a regular grid on VRAM to draw any of these implicit geometries. Custom vertex shaders arrange this grid according to the current geometry type, as depicted in Fig. 2.

[1]Considering IEEE 754 single-precision binary floating-point format.




(a) (b)

Fig. 2. The process executed on vertex shader in order to deform the regular grid to obtain an implicit surface. (a) the regular grid shared between all implicit surfaces. (b) The resulting sphere after deforming (a).

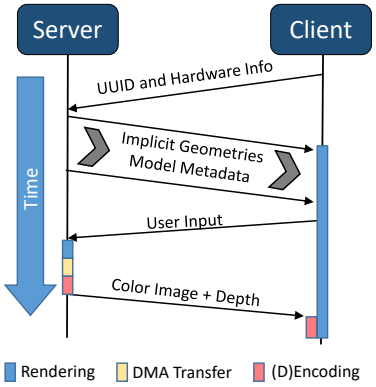



Fig. 3. The communication model between the server and the client. The first frames are displayed after the first geometries arrive on the client side.

We take advantage of hardware-accelerated geometry instancing to render these implicit geometries. Therefore, the system only issues one draw call for each geometry type. This approach vastly reduces the latency on the rendering step, since the issue of too many draw calls is a commonly known bottleneck on drawing large scenes. Previous research has shown that rendering performance can be improved from 6x to 10x by using this approach [20].

### B. *Client-Server Communication*

The first step is to establish a connection between the client and server. At this step, the client informs the server of its hardware capabilities together with the CAD model requisition. If the connection succeeds, the server starts sending the geometries which will be drawn on the client side. In our prototype, the server does a naive geometry selection, sending a fixed amount of geometries to the client depending on the client's capabilities. The first frames are displayed to the user as soon as the first geometries arrive at the client side (Fig. 3). Lastly, the server sends the metadata of individual CAD components. These metadata are contain critical engineering specifications for diverse analysis.

In order to reduce the response delay, the server does not wait for the client to request a new frame at a specific viewpoint to start rendering it. Instead, both client and server

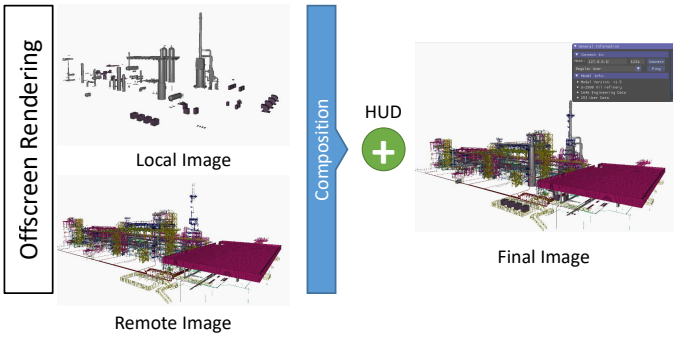



Fig. 4. Schematic representation of the two framebuffers on the client side: one for local rendering and the other for remote rendering. This separation prevents the client to redraw the local scene when a new remote image is available. Instead, the client only combines the local image with the remote image according to their respective depth values. After, the client draws the HUD elements over this combined image.

engines run independently, drawing the scene as long as the user is performing any interaction. The client sends to the server all the user interactions, skipping the repeated ones, along with the world position at which the event took place. The server, on its turn, streams depth-augmented color images to the client along with the corresponding view and projection matrices.

### C. Client Architecture

The client has two threads: the render thread and the resource thread. The render thread is primarily responsible for performing the local rendering. The role of the resource thread is to receive the remote frames, decompress and upload them to the VRAM as textures. In order to obtain a system that is robust to network latency, both threads run independently. The only communication point is when a new remote frame is available, so the resource thread notifies the render thread about it.

In order to prevent redrawing the local scene unnecessarily, the client draws it on a texture. This way, the client has two framebuffers: one for the scene rendered locally and the other for the scene rendered remotely. These two images are combined according to their depth values. Lastly, the client draws the HUD (Heads-up display) elements over this composed image, resulting in the final image to be displayed to the user (Fig. 4). The HUD consists of textual elements, especially the model's engineering metadata. It is necessary for the rendering of these textual elements to take place on the client side, since a lossy compression from the server would severely compromise their readability.

If the viewpoint of the available remote image is different from the client's current viewpoint, a DIBR operation is performed in order to transform the remote image to the current camera position. In this case, when the user stops interacting with the scene, a final remote image will arrive, in which the local and remote viewpoint match. Thereby, the DIBR can be avoided, producing a better-looking final image and preventing the artifacts that arise from the DIBR usage.

The DIBR operation re-projects each point to the world-coordinate system and then projects them back to screen space, but now considering the client's viewpoint. GPGPU technologies, e.g. OpenCL or Compute Shaders, enable this process to be done efficiently. However, the DIBR is done on the client side and our goal is to design a system compatible with outdated and/or limited hardware. Therefore, we devised a method to execute the DIBR operation within the traditional OpenGL pipeline. Our approach creates a point cloud from the remote image. Equation 4 is evaluated for each of these points on the vertex shader, where the $\mathbf{V}$ and $\mathbf{P}$ are the view and projection matrices and the subscript letter indicates the client (c) or server (s). We then compute the final position of each point using the correct matrix transformations. We perform a pixel splat [21] to attenuate any DIBR artifacts.

$$Point_{pos} = \mathbf{P}_c \cdot \mathbf{V}_c \cdot \mathbf{V}_s^{-1} \cdot \mathbf{P}_s^{-1} \cdot Point_{NDC} \qquad (4)$$

### D. Server Architecture

For each incoming connection, the server launches a new render engine thread. A shared bus is responsible for the communication between the clients and their underlying engine instance.

Considering a single engine thread, a new frame is generated as long as its user event list is not empty. As soon as a frame is rendered, the server starts a Direct Memory Access (DMA) transfer in order to capture the frame from VRAM and, afterwards, the rendering of the next frame is started. The DMA transfer prevents locking the engine thread waiting for the synchronization between the CPU and GPU, reducing the latency.

Once the rendered frame is captured, we encode the color and depth buffers. The depth buffer is compressed using the lz4 algorithm [22] because it offers high compression ratio in real time. The color image is encoded with H.264 codec [23]. This codec supports three different types of frames: *i-frame*, *b-frame*, *p-frame*. Among these frames, the b-frames are the ones that increases latency on the system, because they can only be decoded after accessing future frames. Moreover, in our tests they did not reduce the file size significantly. Thus, we do not make use of this type of frame. After the compression of both depth and color image, the server transmits them to the client together with the transformation matrices.

As discussed before, both client and server run independently in order to reduce the system's response delay. Hence, they do not halt their execution waiting the other part to receive a transmitted message. However, this creates a new challenge since now the server must predict where the client viewpoint will be in the next frame. We overcome this shortcoming using two approaches depicted in Fig. 5: *lag compensation* and *path correction*.

Lag compensation attempts to predict where the client camera will be when the remote image arrives on the client side. We use this estimate to displace the server camera in a future position. This way, when the client uses this frame on DIBR operation, more points can be used to produce the

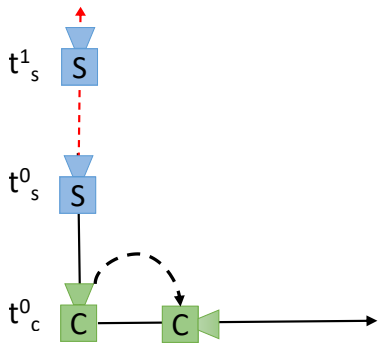



Fig. 5. Schematic representation of the lag compensation and path correction. At $t^0$ the server camera (the blue) is displaced relative to the client one (the green). This displacement in time as being the moving average ($ma$) of the response delay, i.e. $t_s^0 = ma(RD) + t_c^0$. At $t_c^0$, the user rotates the camera to the right and, after this moment, the server camera will be on invalid path (red path). The server is notified only at $t_s^1$ due to the uplink latency, i.e. $upl = t_s^1 - t_s^0$. So, it's necessary to move the server camera back to the position at the event took place, next we execute the user event and lastly we displace the server camera again.

final image, reducing the artifact from the DIBR operation and, consequently, improving the image quality.

However, when the user moves the camera to another direction, the server camera will be at an invalid position until it is notified about the user's event. That is when path correction comes into play: When the event arrives, the server first repositions its camera to the location where the event took place before processing the new user event. That is why the client needs to send the user events along with its current world position.

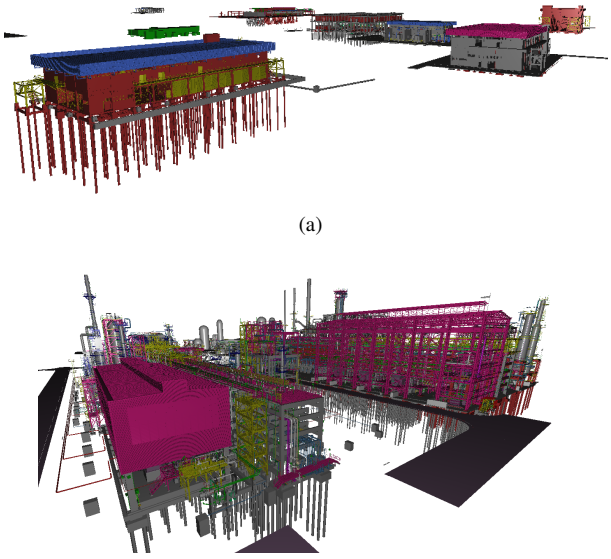
## V. Results

We evaluated our hybrid rendering system with a prototype developed in C++ and OpenGL. Overall system performance was measured using a controlled test environment. Two computers, one for the server and the other for the client, were connected to the same 10 Gb/s local area network (LAN). The server ran on a desktop PC with an Intel Core i7 2.7GHz quad-core processor, 8GB of RAM and a Nvidia GeForce GTX 580 2GB graphics card. The client consisted of a laptop with Intel Core i7 2.6 GHz, 8 GB of RAM and Intel HD Graphics 4000.

The following subsections describe the conducted experiments. First, we measure the advantages of the implicit geometry representations in terms of storage and rendering efficiency. Second, we evaluate overall system performance and discuss the latency involved in each step in both client and server. Third, we analyze the quality of the image produced by our approach.

### A. Network Transmission and Rendering Efficiency

We used two CAD models to make evident possible advantages of using the implicit representation with regard to rendering performance and storage. These two models differ in size and complexity, thus we denote them as *small model* and *large model* (see Fig. 6). Considering only triangular meshes, the *small model* has 144,262 objects and requires 259.4 MB



(a)

(b)

Fig. 6. Two industrial plant models used during our tests: (a) is the *small model* and the (b) is the *large model*.

of VRAM, and the *large model* has 1,105,507 objects and requires 1.9 GB of VRAM.

The implicit representation vastly reduces storage requirements. Considering the implicit representation of these two models, their original size is reduced by 58.1% and 69.7% for the *small* and *large* model, respectively. For the *small model*, only 16% objects remain represented as triangular meshes, since these objects could not be represented by any type described in Table II. For the *large model*, 15.2% of the objects remained as meshes.

The main advantage of this reduction for our system is to transmit object data to the client efficiently. For example, to send all 78,438 parallelograms from the *small model* to the client, we need to transfer 2.69 MB approximately. Considering the IEEE 802.11g Wi-Fi standard, this can be accomplished in less than a second.

When it comes to rendering performance, we compared the OpenGL instanced rendering commands to its traditional rendering commands (*glDrawElements*). For this, we rendered 150,000 implicitly represented objects in our server. The instanced API sped up the rendering to 8.1x, reaching 81 *fps*.

### B. Overall System Performance

To evaluate our system performance in terms of *fps*, we performed rendering tests using three different scenarios:

- 150,000 implicit objects rendered only on the server;
- 150,000 implicit objects rendered only on the client;
- Our hybrid approach, rendering 135,000 implicit objects on the server and 15,000 implicit objects on the client.

Figure 7 shows the obtained *fps* values in each scenario over time. As expected, best *fps* rates (around 80) were obtained

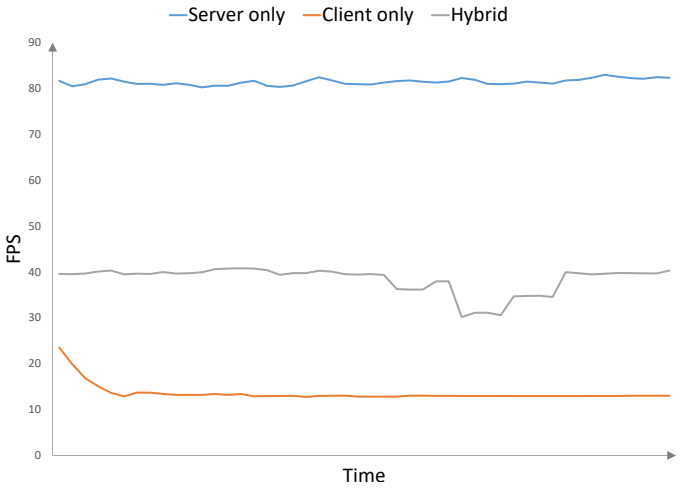



Fig. 7. *Fps* rate obtained using three different approaches: server only; client only and a client-server hybrid rendering. The total execution time in each test is 50 seconds.

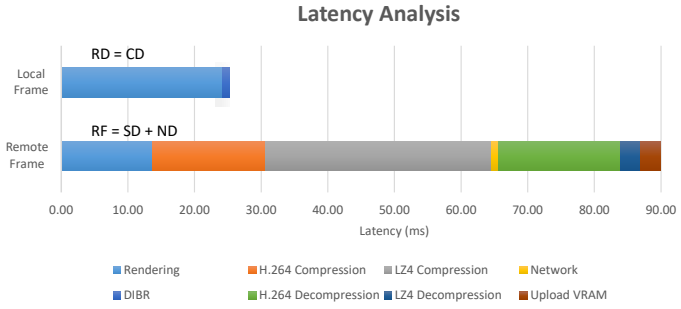



Fig. 8. The latency involved in each step of our methodology. The difference in time between the acquisition of local image and remote image clearly shows the advantage of the hybrid approach. Hence, the user has a faster response even on limited bandwidth.

when we rendered the objects only on the server side, while we had the worst rendering performance on the client-only test (around 13 *fps*). The number of objects rendered on the server or on the client represents a trade-off between performance and final image quality, since locally rendered images suffer no compression. In our hybrid approach test we left 15,000 objects to be rendered on the client, which still resulted in high-quality images being rendered at around 40 *fps*. We evaluate image quality in the next subsection.

Figure 8 demonstrates the advantage of our hybrid methodology over streaming-based remote rendering approaches. Due to the local rendering, our system response delay is about one third of what it would be if the client waited for the server response. In other words, the approach successfully decouples the response delay (RD) from the remote frame delay (RF), delivering interactive rates even when the network is not performing well (a critical factor for enhancing QoE).

### C. Image Quality

Fig. 6 makes evident the high quality images produced by our rendering system. Nevertheless, we wish to conduct a detailed analysis of any limitations in our approach. To this

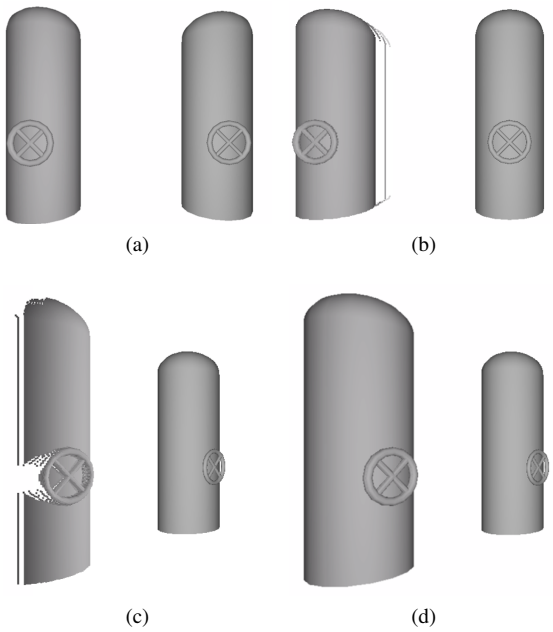



Fig. 9. Image quality comparison during the navigation on the scene. (a) is the initial view. (b) and (c) are the results while navigating through the scene. (d) is the image completion when the user stop interacting with the scene.

end, we isolated the two equipments depicted in Figure 9. The left equipment is rendered on the server whereas the right object is locally rendered. At the client, we captured the image 9a. In this image it is possible to note that the left equipment is blurred when compared to the equipment on the right due to the compression on the server side. The equipment on the left has its details preserved since it was locally rendered.

The Figures 9b and 9c were captured while the user was moving and no new remote frame was available. This figures illustrates the worst scenario, where the network is suffering from high latency. Since we have an outdated remote image, we need to perform DIBR operation on this image. This is why some artifacts emerge in this equipment, on both images, especially on the object's boundaries because the depth values of these regions vary abruptly. In the translational movement (Figure 9b), the remote image still has satisfactory quality. If the user moves away from the object, the image will require less samples than the previous one. However, if the user approaches the object, despite we need more samples to reconstruct the image, the pixel splatting prevents holes on the surface, although the image becomes more blurry.

Nonetheless, in Figure 9c, the remote object becomes incomplete since occluded parts become visible. Despite this, the system preserves the user's spatial notion due to the locally rendered objects and the DIBR rendered objects. Therefore, the user can continue navigating through the scene. As new remote images become available, these regions will become visible and the artifacts disappear over time. When the user finally stops interacting with the system, the client will receive a remote image in which both local and remote viewpoints match. Thus, the final image will contain no artifacts since it

is not necessary to perform a DIBR in this case.

## VI. Conclusion

In this paper, we have presented a novel cloud rendering system that successfully bridged the gap between existing approaches and the specific requirements of massive 3D CAD models. The obtained results showed that our technique can achieve high frame rates with satisfactory image quality even in a constrained environment, such as a high latency network or obsolete computer hardware.

The two key aspects of our work are: (i) the use of implicit geometry representations and (ii) the hybrid rendering mechanism. The proposed architecture matched all requirements established in Section III-C for building a robust and efficient cloud rendering system. The first two requirements, lightweight data transmission and efficient rendering, were accomplished by using implicit representations and instanced rendering. The implicit representation is much more compact than the triangular mesh, reducing the required network bandwidth and the amount of VRAM needed. The final requirement, low response delay, was met by the hybrid approach, since the system is capable of displaying a provisional image while remote rendering results are not available.

The proposed solution paves the way for many research opportunities to improve Quality of Experience (QoE) of cloud rendering systems. First, we wish to develop a prediction model to increase the accuracy of the lag compensation on the server. Predictive techniques could also take advantage of our hybrid rendering approach to improve load balancing between the server and its clients. Different strategies could prioritize which geometries should be rendered locally on the client, depending on observed hardware and network performance. Eventually, a mesh network could fully decentralize graphics processing among servers and clients, akin to a peer-to-peer (P2P) architecture.

## Acknowledgements

This research was supported by CNPq, Conselho Nacional de Desenvolvimento Científico e Tecnológico, grant #154032/2015-8. We thank the reviewers for their helpful comments and suggestions.

## References

[1] B. Reinert, J. Kopf, T. Ritschel, E. Cuervo, D. Chu, and H. P. Seidel, "Proxy-guided Image-based Rendering for Mobile Devices," *Computer Graphics Forum*, vol. 35, no. 7, pp. 353–362, 2016.

[2] R. Shea, J. Liu, E. C.-H. Ngai, and Y. Cui, "Cloud gaming: architecture and performance," *IEEE network*, vol. 27, no. 4, pp. 16–21, 2013.

[3] L. Mui and E. Pearce, *X Window system administrator's guide: for X version 11*. O'Reilly & Associates, Inc., 1992.

[4] B. C. Cumberland, G. Carius, and A. Muir, *Microsoft Windows NT Server 4.0, Terminal Server Edition: Technical Reference*. Microsoft Press Redmond, WA, 1999, vol. 121.

[5] R. A. Baratto, L. N. Kim, and J. Nieh, "Thinc: a virtual display architecture for thin-client computing," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 277–290.

[6] B. K. Schmidt, M. S. Lam, and J. D. Northcutt, "The interactive performance of slim: a stateless, thin-client architecture," in *ACM SIGOPS Operating Systems Review*, vol. 33, no. 5. ACM, 1999, pp. 32–47.

[7] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper, "Virtual network computing," *IEEE Internet Computing*, vol. 2, no. 1, pp. 33–38, 1998.

[8] F. Abraham, W. Celes, R. Cerqueira, and J. L. Campos, "A load-balancing strategy for sort-first distributed rendering," in *Computer Graphics and Image Processing, 2004. Proceedings. 17th Brazilian Symposium on*. IEEE, 2004, pp. 292–299.

[9] T. DeFanti, D. Acevedo, R. Ainsworth, M. Brown, S. Cutchin, G. Dawe, K.-U. Doerr, A. Johnson, C. Knox, R. Kooima *et al.*, "The future of the cave," *Open Engineering*, vol. 1, no. 1, pp. 16–37, 2011.

[10] L. Renambot, A. Rao, R. Singh, B. Jeong, N. Krishnaprasad, V. Vishwanath, V. Chandrasekhar, N. Schwarz, A. Spale, C. Zhang *et al.*, "Sage: the scalable adaptive graphics environment," in *Proceedings of WACE*, vol. 9, no. 23, 2004, pp. 2004–09.

[11] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan, "Wiregl: a scalable graphics system for clusters," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM, 2001, pp. 129–140.

[12] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski, "Chromium: a stream-processing framework for interactive rendering on clusters," *ACM transactions on graphics (TOG)*, vol. 21, no. 3, pp. 693–702, 2002.

[13] F. Lamberti and A. Sanna, "A streaming-based solution for remote visualization of 3d graphics on mobile devices," *IEEE transactions on visualization and computer graphics*, vol. 13, no. 2, 2007.

[14] G. Tamm and J. Krüger, "Hybrid rendering with scheduling under uncertainty," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 5, pp. 767–780, 2014.

[15] X. Liao, L. Lin, G. Tan, H. Jin, X. Yang, W. Zhang, and B. Li, "Liverender: A cloud gaming system based on compressed graphics streaming," *IEEE/ACM Transactions on Networking*, vol. 24, no. 4, pp. 2128–2139, 2016.

[16] C.-Y. Huang, K.-T. Chen, D.-Y. Chen, H.-J. Hsu, and C.-H. Hsu, "Gaminganywhere: The first open source cloud gaming system," *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 10, no. 1s, p. 10, 2014.

[17] G. P. Fickel and C. R. Jung, "Disparity map estimation and view synthesis using temporally adaptive triangular meshes," *Computers & Graphics*, vol. 68, pp. 43–52, 2017.

[18] S. Wang and S. Dey, "Modeling and characterizing user experience in a cloud server based mobile gaming approach," in *Global Telecommunications Conference, 2009. GLOBECOM 2009. IEEE*. IEEE, 2009, pp. 1–7.

[19] M. Claypool and K. Claypool, "Latency and player actions in online games," *Communications of the ACM*, vol. 49, no. 11, pp. 40–45, 2006.

[20] P. I. N. Santos and W. Celes Filho, "Instanced rendering of massive cad models using shape matching," in *Graphics, Patterns and Images (SIBGRAPI), 2014 27th SIBGRAPI Conference on*. IEEE, 2014, pp. 335–342.

[21] W. R. Mark and G. Bishop, "Post-rendering 3 d image warping: visibility, reconstruction, and performance for depth-image warping," Ph.D. dissertation, University of North Carolina at Chapel Hill, 1999.

[22] M. Bartík, S. Ubik, and P. Kubalik, "Lz4 compression algorithm on fpga," in *Electronics, Circuits, and Systems (ICECS), 2015 IEEE International Conference on*. IEEE, 2015, pp. 179–182.

[23] A. Luthra, G. J. Sullivan, and T. Wiegand, "Introduction to the special issue on the h. 264/avc video coding standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 557–559, 2003.