PowerNet: Learning-based Real-time Power-budget Rendering

Yunjin Zhang, Rui Wang, Yuchi Huo, Wei Hua, and Hujun Bao

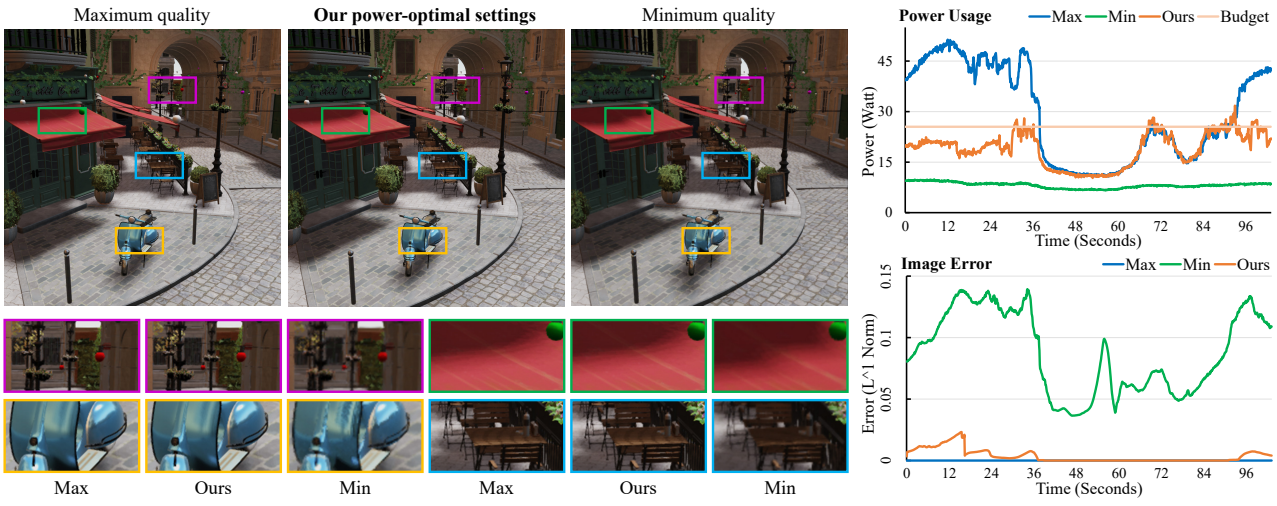

Fig. 1: *We propose a new learning-based power-budget rendering framework that selects the optimal rendering settings to maximize visual quality within a given GPU power budget in real-time. The figure shows the experimental results of the* Bistro *scene, wherein our power-optimal settings drastically reduce power consumption yet produce comparable image quality to the maximum quality setting. The charts on the right show power consumption and quality error (calculated by the pixel-wise L^1 Norm metric).*

Abstract—With the prevalence of embedded GPUs on mobile devices, power-efficient rendering has become a widespread concern for graphics applications. Reducing the power consumption of rendering applications is critical for extending battery life. In this paper, we present a new real-time power-budget rendering system to meet this need by selecting the optimal rendering settings that maximize visual quality for each frame under a given power budget. Our method utilizes two independent neural networks trained entirely by synthesized datasets to predict power consumption and image quality under various workloads. This approach spares time-consuming precomputation or runtime periodic refitting and additional error computation. We evaluate the performance of the proposed framework on different platforms, two desktop PCs and two smartphones. Results show that compared to the previous state of the art, our system has less overhead and better flexibility. Existing rendering engines can integrate our system with negligible costs.

Index Terms—Power-budget rendering, rendering system, neural network

1 Introduction

With the popularity of embedded GPUs in battery-powered mobile devices, real-time rendering technologies have been widely used in various mobile applications, such as mobile games and virtual reality applications. The rendering quality of these applications and the screen resolution of mobile devices [1] are also increasing to meet user preferences.

Given the currently limited progress of commercial battery techniques, people pay many endeavors to extend the durability of mobile devices. For example, real-time rendering applications consume a massive amount of energy and drastically shorten battery life, severely damaging the user experience.

In light of this background, reducing the energy consumption of rendering applications is an urgent need and an important challenge in computer graphics. Recently, many researchers have explored the energy-saving problem of real-time rendering applications [2], [3], [4], [5], [6]. They believe that reducing the amount of computation that has less impact on visual experience is an effective solution.

Among these explorations, power-budget rendering is an under-explored topic [4], [6] but catches extensive attention in the industry. Wang et al. [4] implemented an offline power-budget rendering framework requiring time-consuming initialization, while Zhang et al. [6] presented an online version using heuristic power and quality estimation

• *Authors were in State key laboratory of CAD&CG, Zhejiang University, China. Corresponding author: Rui Wang, rwang@cad.zju.edu.cn*

models. Both of them achieve a good trade-off between power consumption and visual quality. Still, the initialization and the models employed in these two methods heavily rely on view- and scene-related power and quality measurements at runtime. Therefore, these methods require precomputation for new scenes or time-consuming periodic refitting and additional error computation for new views, resulting in low accuracy of power and quality predictions as well as low adaptivity of changes of scenes and views.

In this paper, we design a new self-tuning rendering framework with faster and more accurate predictions to better balance the rendering quality and the power consumption. This framework presents maximized rendering quality on a power budget by automatically selecting the optimal rendering settings for each frame. Our framework requires users to provide a power budget as input; subsequently, all operations are transparent to users. Specifically, we:

- Propose a neural network model utilizes GPU's operating frequencies and rendering settings to predict the power consumption of GPUs;
- Present a neural network model that is capable of predicting the rendering qualities from rendering settings and the occupied image percentages of rendering effects;
- Train these neural network models with data entirely from synthesized scenes, thus significantly reduce the difficulty of preparing datasets;
- Develop a new real-time power-budget rendering framework with proposed power and quality models, enabling self-tuning of rendering parameters for *each* frame to minimize quality error within a given power budget.

We use an in-house OpenGL-based rendering system to implement our prototypical framework. Results from four test scenes show that our method can continuously keep the GPU power consumption below the given budget. It saves 54% relative power consumption at the expense of 21% relative quality error on average (Figure 9).

The remaining part of this paper is organized as follows. Section 2 describes the related work, Sections 3 and 4 define the problem that we are attempting to solve and propose the algorithm overview, respectively. Sections 5 and 6 discuss the structure and the training of our power and quality prediction models. Section 7 describes the approach of selecting the optimal rendering settings. Section 8 details the implementation, and Section 9 shows some of our results. The last section discusses and summarizes the paper.

2 Related Work

Rendering contains a wide range of studies. Offline rendering is one kind of quality-oriented techniques that pursues physically realistic visual results and usually costs minutes to hours to render a single image [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17]. On the other hand, real-time rendering aims to achieve interactive frame rates for performance-oriented applications [18], [19], [20], [21], [22], [23], [24]. In recent years, energy conservation has become an important open issue and inspired a new rendering topic, power-budget rendering. This topic includes both hardware- and software-based solutions, such as energy-saving displays [25], [26], dynamic voltage and frequency scaling (DVFS) [27], [28], variable-precision rendering [29], variable-rate shading [30], [31], and tile-based rendering [32], [33]. Other research has explored new directions, such as GPU power modeling [34], [35] and rendering with dynamic settings [4], [6]. Given that our method shares technologies with rendering utilizing neural networks, we first review rendering methods with neural networks briefly. Then, we discuss previous work on energy-saving rendering, which is highly relevant to the proposed method.

Neural networks in rendering. Neural networks have been successfully used in various rendering applications, such as denoising [36], [37], material modeling [38], [39], [40], material synthesis [41], screen space shading [42], and global illumination [43], to name a few. While these approaches take images or material features as input, our method takes the GPU's statistics at runtime as input to predict power consumption and rendering quality. Such predictions enable a self-tuning power-budget rendering system.

Energy-saving rendering. The DVFS algorithm is one of the most extensively used energy-saving algorithms in integrated circuits and rendering [27], [28]. The key idea of the DVFS algorithm is to dynamically adjust operating frequency and the corresponding voltage in accordance with the current workload. The power consumption of an integrated circuit can be modeled as follows [44]:

$$P = CfV^2 + P_{static}, \tag{1}$$

where P is the power consumed by the integrated circuit; C denotes the capacitance of transistor gates; f represents the operating frequency; and V is the applied voltage governed by f. P_{static} denotes the static power caused by the leakage current. Given that the operating frequency determines the required voltage for keeping transistors stable, the consumed power increases as the cubic of the frequency. Therefore the DVFS algorithm can achieve high energy efficiency by quickly adjusting the operating frequency. Most modern GPUs have integrated the DVFS algorithm on the hardware level; however, this approach does not reduce the computation of rendering.

From a high-level perspective, GPU power can be modeled by considering the scene information. Vatjus-Anttila et al. [34] proposed a power consumption model based on the number of draw calls, triangles, and texels. Huang et al. [35] considered the additional factors of embedded GPUs, such as occlusion, tiles, the number of shading, and the execution time of shaders. However, these proposed models lack usability for multipass rendering frameworks. Zhang et al. [6] have constructed a power model for energy-saving rendering with increased accuracy by considering different rendering processes, such as shadows, reflections, and anti-aliasing. Generally, these high-level models can predict the power consumption of any given 3D scene without considering the characteristics of the underlying hardware.

Dynamic rendering setting is an under-explored area in the literature. Wang et al. [4] provided a straightforward power-budget rendering framework requiring heavy scene- and device-related precomputations. Zhang et al. [6] proposed an online optimization system, which saves a lot of time in precomputation but still needs time-consuming

periodic refitting and additional error computation for new views. In sum, neither of them is free from view- and scene-dependent power and quality measurements, resulting in low accuracy and adaptivity in new scenes and views.

The problem is, if the workload changes drastically, the GPU's operating frequency fluctuates considerably due to the DVFS algorithm. Thus the accuracy of the power predictions based on the linear regression model [6] decreases. As a result, this model has to be periodically refitted, which leads to optimization gaps in the real-time rendering. Moreover, fitting this model requires real-time access to power measurements, which is nearly impossible on many mobile devices (e.g., Android smartphones without root permission).

Besides, the latest method [6] has to compute the quality errors of each rendering pass to select the optimal rendering configurations, which causes additional overheads at runtime. Additionally, it does not reuse any information among frames. It just selects the optimal configuration for one frame and then waits until the next trigger comes.

By contrast, we exploit a neural network model to predict power consumption under various operating frequencies during rendering. This network model is view- and scene-independent and produces high-accuracy predictions for any scene on many hardware platforms without the need for power measurement at runtime. Moreover, we apply a view- and scene-independent quality prediction model in predicting the quality errors of different rendering settings. This quality model reduces the additional computation and reuses the information, supporting selecting the optimal settings for each frame.

3 Problem Definition

We regard the real-time rendering process as a set of rendering effects organized in a particular order that produce different visual effects. In this manner, an image rendering function f can be abstracted from the entire rendering process by taking two main parameters $(\mathbf{s}, \mathbf{c})$ as input to produce the final color image.

- Rendering setting $\mathbf{s}$: It defines the rendering effects (e.g., lighting, shadow mapping, and anti-aliasing) required for executing function f, as well as the visual quality settings selected for different effects (e.g., lighting model, shadow map resolution, and searching steps of anti-aliasing). In our implementation, $\mathbf{s}$ is a settings vector that each dimension represents the quality level of a rendering effect.
- Camera parameter $\mathbf{c}$: It defines the current position and view of the camera.

Generalizing the rendering processes to function f implicitly includes forward and deferred rendering frameworks. Technically, given the same camera parameter $\mathbf{c}$, different settings $\mathbf{s}$ produce different images from function f. Let $\mathbf{s}_{\text{best}}$ be the rendering setting that generates the best-quality image. Similar to a previous work [4], we define the image quality error $e(\mathbf{c}, \mathbf{s})$ of rendered by settings $\mathbf{s}$ from camera $\mathbf{c}$ as

$$e(\mathbf{c}, \mathbf{s}) = \int\int_{uv} ||f(\mathbf{c}, \mathbf{s}_{\text{best}}) - f(\mathbf{c}, \mathbf{s})|| du v, \qquad (2)$$

TABLE 1: *Main symbols used in the paper.*

$\mathbf{s}$	Rendering setting, a vector with quality levels for each effect
s^i	Quality level for effect i
$\mathbf{s}_{\text{best}}$	Rendering setting that produces the best quality images
$\mathbf{c}$	Camera parameter, including position and view
$f(\mathbf{c}, \mathbf{s})$	Image rendering function with $\mathbf{c}$ and $\mathbf{s}$
$e(\mathbf{c}, \mathbf{s})$	Image quality error of $f(\mathbf{c}, \mathbf{s})$
$P(\mathbf{c}, \mathbf{s})$	GPU power consumption of $f(\mathbf{c}, \mathbf{s})$
P_{bgt}	GPU power budget
$\mathbf{d}$, $\mathbf{v}$, $\mathbf{f}$	Vectors of draw calls, vertices, and fragments of all rendering effects of $f(\mathbf{c}, \mathbf{s})$
$\mathbf{a}$	Vector of percentage occupied areas of all rendering effects of $f(\mathbf{c}, \mathbf{s})$

where u and v represent the 2D pixel coordinates of the image, and $||\cdot||$ indicates a chosen error metric.

Different $\mathbf{c}$ and $\mathbf{s}$ also produce different energy consumption, which we define as $P(\mathbf{c}, \mathbf{s})$. We tested various $f(\mathbf{c}, \mathbf{s})$ and found that rendering the minimum quality images can reduce power consumption to 16.8% (i.e., save 83.2% relative power consumption) of rendering the maximum quality images. Generally, rendering high-quality images requires high energy consumption, and reducing the energy of rendering processes increases image artifacts. It generates a vast two-dimensional space of trade-offs between energy consumption and image quality. Given a power budget P_{bgt}, we attempt to find the optimal $\mathbf{s}$ that minimizes $e(\mathbf{c}, \mathbf{s})$ and maintains $P(\mathbf{c}, \mathbf{s})$ within the budget, as shown as follows:

$$\mathbf{s} = \arg\min_{\mathbf{s}} e(\mathbf{c}, \mathbf{s}) \quad \text{subject to} \quad P(\mathbf{c}, \mathbf{s}) < P_{\text{bgt}}. \qquad (3)$$

We call this process power-budget rendering. Different from the state-of-the-art work [6], we use a network model to predict power consumption at each frame, allowing us to avoid 1) sampling power measurements, 2) refitting the power prediction model, and 3) checking the model's accuracy during rendering. We also use an efficient network model in sorting $e(\mathbf{c}, \mathbf{s})$ to considerably reduce the computational overhead of the quality error estimation proposed by the previous work. As a result, we spend considerably less precomputation time than Wang et al.'s work [4] (mainly used to generate the training dataset) but achieve better flexibility and effectiveness than previous work [4], [6]. Table 1 summarizes the symbols used in this paper.

4 Algorithm Overview

The searching of optimal rendering settings in our power-budget rendering framework is a multiobjective optimization problem in the two-dimensional energy-quality space. Figure 2 shows an overview of our algorithm.

In the neural network training phase, we train the power and quality prediction models using rendering statistics, including scene complexity, rendering settings, GPU power and operating frequency, and quality errors, as well as scene information. However, we do not use any data from real scenes for model training. Thus, the training dataset is entirely generated using dummy scenes, which makes our models view- and scene-independent.

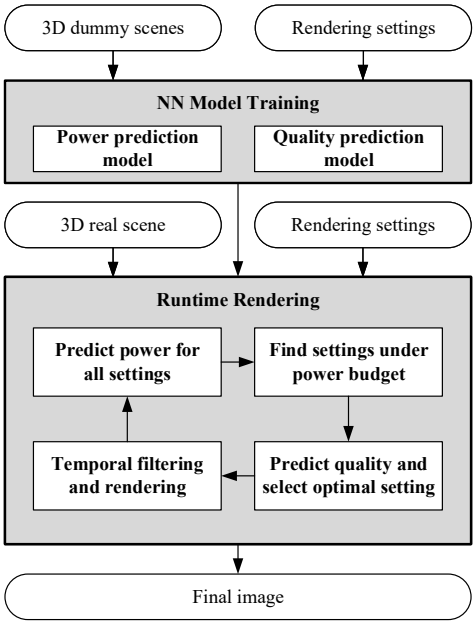

Fig. 2: *Algorithm overview. Our algorithm is divided into two phases, namely, the neural network training phase (see Sections 5 and 6) and the following runtime rendering phase (see Section 7).*

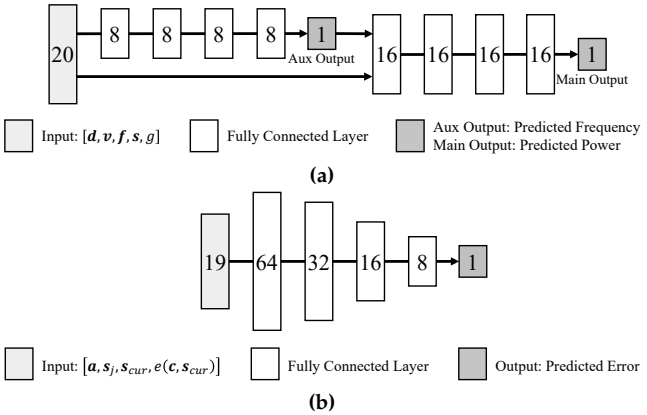

Fig. 3: (**a**) *Network structure of our power prediction model. The input is a vector containing* $\mathbf{d}$, $\mathbf{v}$, $\mathbf{f}$, $\mathbf{s}$, *and* g; *this vector is used to represent the state* $(\mathbf{c}, \mathbf{s}, g)$. *The auxiliary output is a scalar denoting the predicted GPU operating frequency, which is an input of the subsequent subnetwork. The main output is a scalar representing the predicted GPU power consumption of the given state. We train the two subnetworks simultaneously.* (**b**) *Network structure of our quality prediction model. The input is a vector containing* $\mathbf{a}$, $\mathbf{s}_j$, $\mathbf{s}_{cur}$, *and* $e(\mathbf{c}, \mathbf{s}_{cur})$. *The output is a scalar that represents the predicted quality error of the given state* $(\mathbf{c}, \mathbf{s}_j)$.

In the runtime rendering phase, when users freely explore the scene, we first use the power prediction model to predict the power consumption of all the rendering settings periodically and find those settings below the given power budget. Then, we select the optimal rendering settings on the basis of the quality prediction model to minimizes the quality error and use it to render the final image.

The following sections detail these two phases. We assume that the developers can obtain the real-time power measurements of GPUs or mobile devices, as well as the operating frequencies of GPUs, in the training phase. During runtime rendering, we only need the users to specify a power budget, and the rest is automatic and transparent.

5 Neural Network Model for Power Prediction

This section defines the power function and describes the network structure and the training of the power prediction model correspondingly. Once the model is trained, we can integrate it into the rendering system in the runtime rendering phase (Section 7).

5.1 Power Function

In a real-time rendering pipeline, when we issue an instruction to the GPU to draw a list of triangles, such an instruction is called a *draw call*. These triangles then go through a series of stages. First, the *vertex shader* executes operations on each vertex. Then, the rasterization performs clipping and culling to discard the invisible parts and converts the vertex data to fragments. Finally, the *fragment shader* runs a series of instructions to determine the final color for each pixel.

When the frame rate stays constant, an increase in scene complexity results in the increment in computational complexity (e.g., more draw calls, shader invocations, and code complexity of shaders) and power consumption. Besides, multifarious rendering effects and their various quality levels also cause a vast space of computational complexity. For example, setting a different number of samples of screen space reflections results in different computational overheads. In addition, the varieties of camera parameters $\mathbf{c}$ implicitly express different computational complexities. On this basis, we quantify all the rendering effects into computational complexity and formulate the power function Φ as

$$P(\mathbf{c}, \mathbf{s}, g) = \Phi(\mathbf{d}, \mathbf{v}, \mathbf{f}, \mathbf{s}, g), \tag{4}$$

where $P(\mathbf{c}, \mathbf{s}, g)$ denotes the power consumption of the GPU g at the camera parameter $\mathbf{c}$ using the rendering setting $\mathbf{s}$. Similar to the previous work [6], $\mathbf{d}$, $\mathbf{v}$, $\mathbf{f}$, and $\mathbf{s}$ are vectors representing *draw calls*, *vertex*, *fragments*, and rendering effects respectively. The tuple $(\mathbf{d}^i, \mathbf{v}^i, \mathbf{f}^i, \mathbf{s}^i)$ denotes the numbers of draw calls, vertex invocations, fragment invocations, and the rendering level setting for the ith rendering effect. Using this representation, we can implicitly represent the dynamically changing camera parameter $\mathbf{c}$ with $\mathbf{d}$, $\mathbf{v}$, and $\mathbf{f}$ by counting the visible objects and rendering effects in the scene. Generally, given a rendering setting $\mathbf{s}$, a camera parameter $\mathbf{c}$, and a specific GPU g, we can obtain all the inputs of the power function Φ, and then compute the power consumption corresponding to the state $(\mathbf{c}, \mathbf{s}, g)$.

5.2 Network Architecture

We design a neural network model to learn the implicit power function Φ. This model considers the effect of GPU operating frequency under various workloads for power prediction, as shown in Figure 3a. This model contains two subnetworks. The first one consists of four hidden fully-connected (FC) layers, and each layer has eight nodes. This subnetwork takes the state $(\mathbf{c}, \mathbf{s}, g)$ as the input to predict the corresponding GPU operating frequency. In the implementation, the input dimension is 20 (see subsection 8.3 and Table 2 for the details). The second subnetwork is also

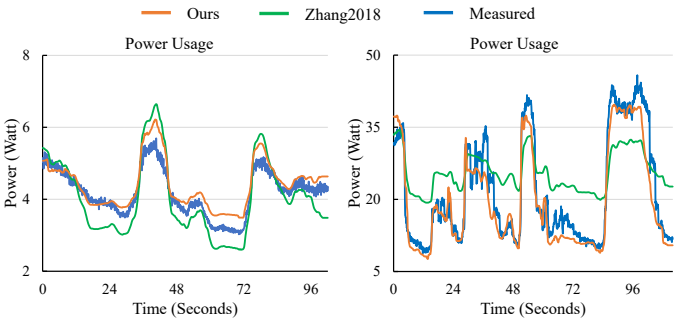

Fig. 4: *Measured power consumption and corresponding predictions of the* Island *(left) and the* San Miguel *(right) scenes. Although the two models work well in the simple case, the linear regression model [6] is less accurate than ours when the scene complexity changes rapidly.*

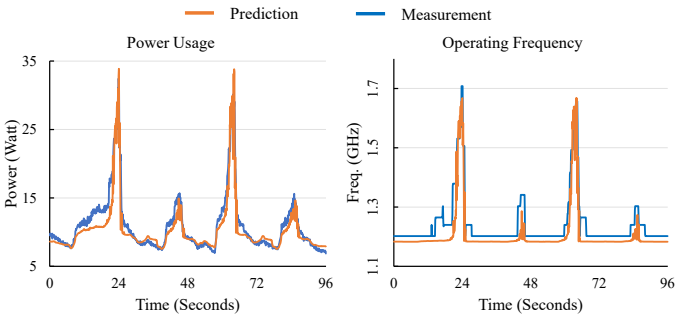

Fig. 5: *Power and frequency predictions of the* Subway *scene from the power prediction model and corresponding measurements. The scene is rendered by NVIDIA Quadro P4000.* ***Left****: Power usages.* ***Right****: Operating frequencies.*

composed of four hidden FC layers, but each layer has 16 nodes. This subnetwork takes the predicted frequency from the previous subnetwork and the state $(\mathbf{c}, \mathbf{s}, g)$ as inputs to predict the final GPU power consumption. All of the FC layers in the two subnetworks use the *softplus* [45] activation function.

One notable feature of the network model is that we explicitly predict the GPU frequency as an input of the second subnetwork. The state-of-the-art work [6] did not consider the operating frequency of GPU, but we found it helpful to the prediction accuracy. While the GPU can provide a stable frame rate without increasing its operating frequency in a simple scene, it has to increase the operating frequency to stabilize the frame rate in a complex scene, which results in a significantly increased power consumption, in accordance with the power model of integrated circuits (Equation (1)).

Figure 4 depicts the power predictions and measured power consumption of two scenes (i.e., *Island* and *San Miguel* scenes) running with the same GPU. In the *San Miguel* scenario, operating frequency varies significantly due to the high computational complexity at some viewports; this variation ultimately results in a wild fluctuation in power consumption. In that case, the linear regression model [6] produces less accurate power predictions and needs to perform refitting processes at some periods, which leads to less optimal rendering settings.

5.3 Network Training

In order to train the aforementioned network model, the first step is to generate the training set. Even though using data from real scenes can significantly improve the accuracy of our power prediction network for the corresponding scenes, in order to enhance the flexibility of our model, we only use dummy scenes (see the supplementary materials for examples) to generate the training set on different GPUs. We progressively increase the complexity of dummy scenes with uniformly sampled settings $\mathbf{s}$. Then, we collect the number of invocations of pipeline stages (i.e., *draw calls*, *vertex*, and *fragments*) and measure the power consumption and the operating frequencies of GPUs during rendering. We collect more than $100,000$ training data on four different GPUs, wherein each data contains a vector of input and two scalars of output (frequency and power). It takes us less than two days to generate these synthetic data.

After generating the training set, the second step is to train the network model. Given the considerable differences in computational power and energy consumption between the mobile and the desktop platforms, we decide to train the network separately for the two platforms. However, to improve flexibility, we train an identical network for different GPUs on the same platform. The training set mixes data from different types of GPUs to let the network learn different patterns of power consumption on different GPUs. Once trained for a platform, no precomputation is needed for new scenes and views on involved GPUs of this platform. On new GPUs, our network also exhibits a certain degree of transferability across similar computation patterns, by abstracting some of the dependencies of hardware (see Section F of the supplementary document for details). Specifically, the predicted power consumption of our network is proportional to the measured one. However, our network still needs a retrain to accurately predict power for new GPUs.

We use the measured operating frequencies and power consumption of GPUs to supervise the first and second subnetworks, respectively. For the first subnetwork, we use the mean absolute percentage error (MAPE) as the loss function during training. The second subnetwork takes the $logcosh$ as the loss function. We use the stochastic gradient descent (SGD) algorithm as the network optimizer with a learning rate of 0.0003 to train the two subnetworks simultaneously, spending approximately 15 epochs (which takes less than 1 minute) to converge. The validation set (a subset of the original training set) reports that the MAPEs of predicted power and frequency are less than 4% and 3%, respectively. Figure 5 shows some predictions.

6 Neural Network Model for Quality Prediction

This section describes the quality prediction model of our real-time power-budget rendering framework. This model shares training scenes with our power prediction model, but has different training data and architecture.

6.1 Quality Function

In addition to the vast space of computational complexity, various rendering settings usually produce different rendering qualities, forming an enormous space of quality errors. It is a very challenging work to quickly select high-quality rendering settings at a small cost. Previous methods [4],

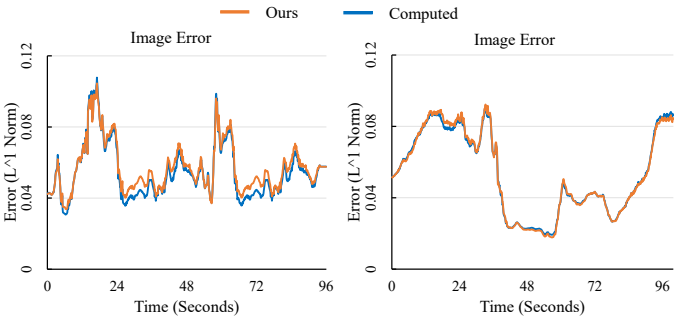

Fig. 6: *Computed quality errors and corresponding predictions* ***Left:*** Subway *scene with a rendering setting* $\mathbf{s} = (max, min, min, min, min, mid)$. ***Right:*** Bistro *scene with a rendering setting* $\mathbf{s} = (min, max, max, max, max, max)$.

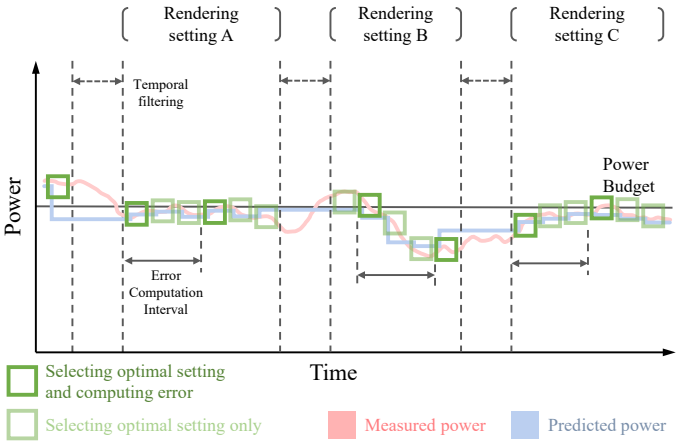

Fig. 7: *Timeline illustrating power usage during rendering, and how our algorithm works. Given an Error Computation Interval, our algorithm computes and stores the quality error of current frame periodically (dark green box). At each frame except those of the temporal filtering, we select the optimal rendering setting* $\mathbf{s}_{opt}$ *on the basis of the latest quality error (dark green and light green box). Our algorithm reuses the previous quality error when the current frame has not performed an error computation. If the new setting is different from the old one, then our algorithm performs temporal filtering to transition to the new setting smoothly. Please refer to the text for details.*

[6] have attempted to entirely and accurately obtain accurate quality errors $e(\mathbf{c}, \mathbf{s})$, but have the disadvantages of requiring long preprocessing time or heavy computational overheads. Instead, we propose a new quality function Ψ to express a mapping between the quality errors of any two rendering settings. Thus, it can calculate quality errors of all other settings at the same frame through the error of the current setting.

Specifically, this function takes scene information $\mathbf{a}$, rendering setting $\mathbf{s}_j$, and current rendering setting $\mathbf{s}_{cur}$ and its image error $e(\mathbf{c}, \mathbf{s}_{cur})$ as inputs to predict the error $e(\mathbf{c}, \mathbf{s}_j)$, which can be formulated as follows:

$$e(\mathbf{c}, \mathbf{s}_j) = \Psi\left(\mathbf{a}, \mathbf{s}_j, \mathbf{s}_{cur}, e(\mathbf{c}, \mathbf{s}_{cur})\right), \tag{5}$$

where $e(\mathbf{c}, \mathbf{s}_j)$ represents the quality error at the camera parameter $\mathbf{c}$ using the rendering setting $\mathbf{s}_j$. Scene information $\mathbf{a}$ is a vector denoting the percentage occupied areas of all rendering effects. Each dimension of $\mathbf{a}$ represents how much percentage area an effect occupies in the current viewport $\mathbf{c}$.

6.2 Network Architecture and Training

The image quality function Ψ is highly nonlinear and hard to compute. Therefore, we design a neural network model to express function Ψ approximately. A visualization of the architecture of this model is shown in Figure 3b. In the implementation, the input dimension is 19 (see subsection 8.3 and Table 2). This model contains four hidden FC layers. The first layer has 64 nodes, and each subsequent layer is half of the previous layer. All the layers use the *softplus* activation function.

The generalization across various viewports and scenes is an essential feature of the quality prediction model. Because the model directly maps the quality errors between different rendering settings at the same camera parameter $\mathbf{c}$, the problem is therefore significantly decoupled from the changes of viewports and scenes. It narrows down the problem space and makes the training easier. Given that the mapping is conducted at the same $\mathbf{c}$ that leads to the same image structure, we use a relative pixel-wise error metric instead of the previously used SSIM algorithm [4], [6] to calculate quality errors for different rendering settings. Particularly, we choose the arithmetic mean of the numerical difference of each pixel, which we call the standardized L^1 norm metric, as shown as follows:

$$e(\mathbf{c}, \mathbf{s}) = L\left(\frac{1}{N}\sum_{i=1}^{N}\left|\frac{P_i^{\mathbf{s}_{\text{best}}} - P_i^{\mathbf{s}}}{L(P_i^{\mathbf{s}_{\text{best}}}) + \varepsilon}\right|\right), \tag{6}$$

where $e(\mathbf{c}, \mathbf{s})$ denotes the quality error of rendering setting $\mathbf{s}$ at the camera parameter $\mathbf{c}$. Function L converts the RGB color to luminance, and N is the number of pixels. $P_i^{\mathbf{s}}$ represents the RGB color of the ith pixel when rendering with $\mathbf{s}$, and $\mathbf{s}_{\text{best}}$ is the best quality rendering setting. ε is a small positive number (we set $\varepsilon = 10^{-3}$). $|\cdot|$ returns the absolute value.

Similar to the training phase of the power prediction model, we train our quality model on a dataset consisting entirely of dummy scenes (see the supplementary materials). In addition to further enhance the generalization of the quality prediction model, we also rotate the camera to increase the variety of training data. We uniformly took samples from 196 rendering settings to generate more than 48M pairwise combinations as the training set. This quality prediction model takes the mean square error as the loss function and the SGD algorithm as the optimizer, with a learning rate of 0.1. The mean absolute error of predicted quality errors is less than 0.005 in our validation set (20% of the original training set).

To evaluate our quality network on real scenes, we compare the predicted quality errors with the computed quality errors, of given rendering settings in two scenes. Most of the time, the quality errors inferred by our quality prediction model are highly close to the computed quality errors. Figure 6 shows some experimental results of the comparison between predicted errors and computed errors.

7 Real-time Power-budget Rendering Framework

In the previous sections, we described the network architectures of the power and quality prediction models and their training processes. We demonstrate that our power prediction model has higher accuracy and flexibility compared with the regression model [6]. Moreover, our quality prediction model has fewer GPU overheads than the previous method. With such trained network models, we can get rid of the time-consuming precomputation for new scenes [4] or the runtime periodic refitting and additional error computation [6]. We now describe how to integrate them into the rendering system to predict power consumption and quality errors and then select the optimal settings within a power budget during real-time rendering. Generally, we select the optimal rendering settings for each frame and perform a temporal filtering process for a smooth transition from the current setting to the new one. Figure 7 illustrates the entire process of our power-budget rendering.

7.1 Selecting Optimal Rendering Setting

Given a scene, a camera parameter $\mathbf{c}$, the power prediction model Φ, and the quality prediction model Ψ, our goal is to find an optimal rendering setting $\mathbf{s}_{opt}$ with the minimum quality error below the power budget, as mentioned in the algorithm overview (Section 4). Algorithm 1 gives the pseudocode of the setting selection process, where S denotes the set of all rendering settings.

Algorithm 1 Selecting the Optimal Rendering Setting $\mathbf{s}_{opt}$

Input: $\mathbf{d}, \mathbf{v}, \mathbf{f}, S, g, P_{\text{bgt}}, \mathbf{a}, \mathbf{s}_{cur}, e(\mathbf{c}, \mathbf{s}_{cur})$
Output: $\mathbf{s}_{opt}$

```
S' = ∅
for each s_i in S do
    P_predict(c, s_i, g) = Φ(d, v, f, s_i, g)
    if P_predict(c, s_i, g) ≤ P_bgt then
        S' = S' ∪ {s_i};
    end if
end for
for each s_i in S' do
    e_predict(c, s_i) = Ψ(a, s_i, s_cur, e(c, s_cur))
end for
Sort S' in ascending order by e_predict(c, s_i)
s_opt = the first element of S'
```

First, we collect the pipeline statistics of the current state $(\mathbf{c}, \mathbf{s}, g)$. Then, for a given power budget, we infer the power consumption of all the rendering settings at the current viewport using our power prediction model and then discard any settings wherein the predicted power exceeds the budget.

Thereafter, we compute and store the quality error of the current frame periodically. The interval during this period is called the Error Computation Interval. Frames without an error computation utilize the previous quality error as the current one. For the remaining settings that meet the power budget, we infer their quality errors using the quality prediction model and then sort them in ascending order on the basis of their errors. Finally, we select the optimal rendering settings with the minimum predicted error among these settings. In this manner, we separate the two-dimensional spatial search into two linear searches, eliminating the need to generate the Pareto frontier in the power-error space.

7.2 Runtime Power-budget Rendering

We have detailed how to find the optimal rendering settings under a given power budget in the previous subsection. Here, we describe how to use them for real-time rendering. At the beginning of rendering, we can instantly select the optimal setting. That is, we can start our power-budget rendering immediately. Subsequently, we perform the selection process for each frame but compute the quality error periodically. To avoid sudden changes in visual quality after having selected a different setting, we perform a temporal filtering process mentioned by Wang et al. [4] to smoothly transit to the newly selected setting. During a time window T for filtering, the currently used setting, $\mathbf{s}_{cur}$, is computed by a temporal interpolation:

$$\mathbf{s}_{cur} = \left[\left(1 - \frac{t}{T}\right) \mathbf{s}_{old} + \frac{t}{T} \mathbf{s}_{new} \right], \tag{7}$$

where the brackets represent the closest integer. $\mathbf{s}_{old}$ and $\mathbf{s}_{new}$ denote the previously and newly selected optimal setting, respectively. t is the time elapsed after applying a new rendering setting. Note that narrowing T may result in sudden changes in visual quality, but expanding T may lead to obsolete selections.

8 Implementation

We have described the details of the power and quality prediction models and the power-budget rendering framework. In practice, we implemented an OpenGL-based rendering system and tested it on four devices. Two of them belong to the desktop platform, one of which is a PC with Intel Core i7-3770K CPU and NVIDIA Quadro P4000 graphics card; the other consists of Intel Core i7-7700 and NVIDIA GeForce RTX 2080Ti. The rest belong to the mobile platform, one of which is a smartphone with an 8-core Kryo CPU and an Adreno 630 GPU; the other has an 8-core Cortex CPU and a Mali-G76 GPU. The PCs were both running on Microsoft Windows 10, and the smartphones were running on Android 9.0.

8.1 Power Measurement

We set the rendering system to a constant frames-per-second rate to avoid the influence of frame rate on power consumption. Thus, the power consumption of GPU depends only on scenes, rendering settings, and camera parameters. For the desktop platform, we fixed the memory frequency of the graphics card to reduce the potential influence. The specific power measurement method varies by the platform.

Desktop Platform: We used the C-based API in NVIDIA Management Library (NVML) [46] to directly measure the instantaneous power of the GPU. The document reports an error of the API within $\pm 5\%$. To reduce variance, we read power back 20 times per second and spent 2 seconds measuring and averaging them for each sample in our dataset. The interval between the power measurements of each sample is 2 seconds to reduce any residual influence.

Mobile Platform: We used the built-in shell commands of the Android system to read back the battery's real-time

TABLE 2: *Lists of parameters used in rendering settings.*

Effects	Parameters	Values (max, mid, min)
Resolution	buffer resolution	100%, 80%, 60%
Reflections	(samples, kernel)	(64,9), (16,1), off
Shadows	map resolution	2048, 1024, 512
Base shading	lighting model	microfacet, Lambert, cheap
Metals	samples	60, 6, 2
Anti-aliasing	presets	good, fast, off

voltage and current values recorded in the system file, and then calculate the instantaneous power. As a result, we obtained the total power of the smartphone instead of the GPU. Thus, we closed all nonessential applications and services before measuring to minimize interference. Nonetheless, power measurements on the mobile platform have a more significant variance than those on the desktop. We read back once per second and averaged over 10 seconds. The interval between the power measurements of each sample is 5 seconds.

8.2 Rendering Framework

We implemented an in-house OpenGL rendering system in C++, which can be easily modified to OpenGL ES and then port to the mobile platform. Our rendering framework supports six configurable rendering effects, including resolution, Screen Space Reflection (SSR), shadow mapping, base shading, metal shading, and Fast Approximate Anti-aliasing (FXAA). Each effect has three different quality levels (i.e., maximum, middle, and minimum) determined by controllable parameters, which amounts to a total of 729 rendering settings. The complete set of parameters and their values is shown in Table 2. We enabled the View Frustum Culling and the Back-face Culling to reduce overheads.

The resolution effect sets the size of buffers, changing the number of fragment invocations of all other effects except the shadow mapping effect. Technically, the resolution effect is not a real rendering pass; however, it has a significant contribution to the power consumption and visual quality. Thus, we treated it as a rendering effect and implemented different levels for convenience. The SSR effect provides reflections for some objects with specular materials. Lowering the quality level reduces the number of ray-tracing steps and the kernel sizes of color filtering [47]. The minimum level turns off this effect. For shadow mapping, we took the shadow map resolution to represent its quality level. This effect only considers the generation of shadow maps, whereas the shadow map lookup is considered in the resolution effect. For base shading, we implement three lighting models. The highest quality is based on the microfacet model, followed by the Lambert shading model, and the minimum quality uses a cheap lighting algorithm. The metal shading effect is a lighting algorithm of GPU-based importance sampling. We varied the number of samples for different quality levels. The FXAA effect is a morphological anti-aliasing algorithm [48]. The good preset has more search steps than the fast preset, and the lowest level turns off this effect.

8.3 Network Inputs

The input of power prediction network is a vector containing draw calls $\mathbf{d}$, vertices $\mathbf{v}$, fragments $\mathbf{f}$, quality settings $\mathbf{s}$, and GPU model g, with a total of 20 dimensions. Specifically, parameters of six rendering effects compose the input vector as: reflections $(\mathbf{d}^1, \mathbf{v}^1, \mathbf{f}^1, \mathbf{s}^1)$, shadows $(\mathbf{d}^2, \mathbf{v}^2, \mathbf{f}^2, \mathbf{s}^2)$, base shading $(\mathbf{d}^3, \mathbf{v}^3, \mathbf{f}^3, \mathbf{s}^3)$, metals $(\mathbf{d}^4, \mathbf{v}^4, \mathbf{f}^4, \mathbf{s}^4)$, anti-aliasing $(\mathbf{f}^5, \mathbf{s}^5)$, and resolution $(\mathbf{s}^6)$. All effects share scalar g (see Equation (4)). In our implementation, the resolution effect acts as a scaling factor without draw calls, vertices, and fragments. The anti-aliasing effect has constant draw calls and vertices. Thus, effects resolution and anti-aliasing have fewer parameters as inputs compared with other effects.

Our quality prediction network takes another input vector with 19 dimensions containing $\mathbf{a}$, $\mathbf{s}_j$, $\mathbf{s}_{cur}$, and $e(\mathbf{c}, \mathbf{s}_{cur})$. For six rendering effects, each effect i ($i = 1, 2, ..., 6$) has a tuple of input parameters $(\mathbf{a}^i, \mathbf{s}^i_j, \mathbf{s}^i_{cur})$, and scalar $e(\mathbf{c}, \mathbf{s}_{cur})$ is shared by all effects (see Equation (5)).

When the number n of rendering effects increases, the number of settings grows exponentially ($O(3^n)$), but input parameters increase linearly ($O(4n + 1)$ for power model and $O(3n + 1)$ for quality model).

Compared with the previous work [6], we did not explicitly use the number of instructions and texel accesses as input parameters but implicitly encode it into quality levels, i.e., shader with long instructions usually has a high-quality level.

8.4 Quality Error Computation

We calculated the per-pixel difference on the basis of Equation (6) between two textures in the fragment shader of the last rendering pass. Then, we generated the 1×1 mipmap (the average difference of all pixels) using GPU to determine the final image error. This method takes an average of $0.5\ ms$ on the desktop platform, which is $100\times$ faster than the previously used SSIM algorithm [4], [6] but with proportional similarity in quality.

8.5 Rendering Statistics

Collecting the number of invocations on the desktop platform by using OpenGL query objects is convenient. On the mobile platform, we had to count the number of vertices from models instead of from the pipeline due to the lack of support from the OpenGL ES APIs. In addition, we rendered a mask to the alpha channel and generated 1×1 mipmaps, which represent the percentages of shaded pixels, to count the fragments. We assume invocations of two consecutive frames are similar so that they are queried in the previous frame and used in the current frame.

Calculating the percentage occupied areas of partial-screen effects (including reflections, shadows, base shading, and metals) is similar to counting the fragments on the mobile platform. For the full-screen effects (i.e., resolution and anti-aliasing), we performed a fast edge detection algorithm [49] based on the luminance of each pixel and generated mipmaps to obtain the percentage area of edges.

8.6 Implementation Details

We perform the network inferences on the CPU to predict the power consumption of all the 729 rendering settings each time, with an average cost of $1.5\ ms$ and $4\ ms$ on the desktop and the mobile platform, respectively. Predicting quality errors and selecting the optimal rendering setting on the CPU takes approximately $0.8\ ms$ and $2.1\ ms$ on the corresponding platforms. Besides, increasing framerates

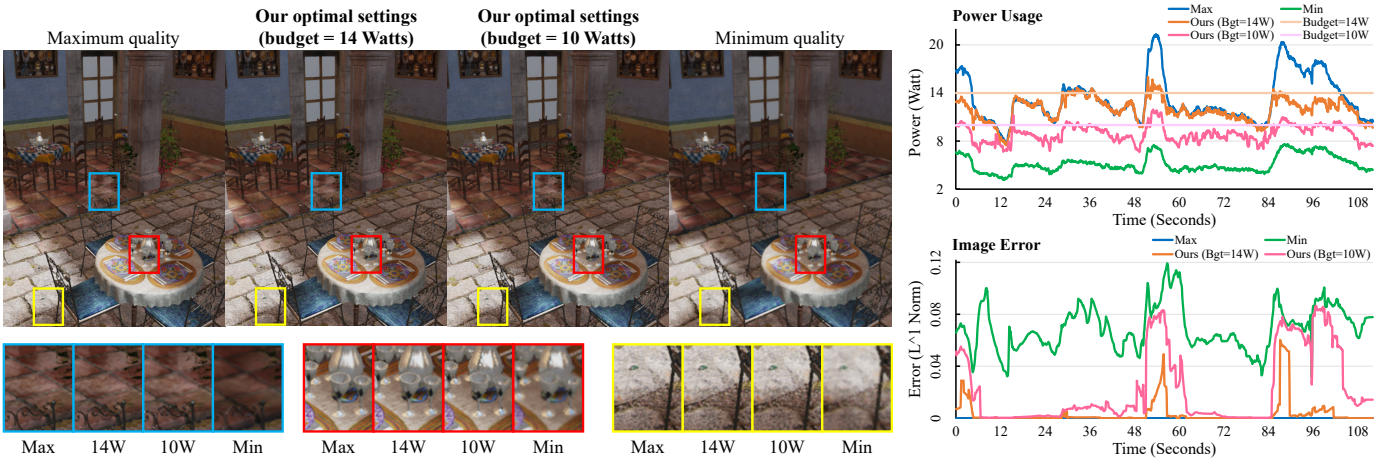

Fig. 8: *Real-time demo for the* San Miguel *scene, executed on a desktop PC with the RTX 2080Ti graphics card. We compare the maximum and minimum quality levels against our power-budget rendering system with different power budgets of* 10 W *and* 14 W*. Images generated by our method under the higher budget are nearly indistinguishable from those produced with the maximum quality, whereas energy savings achieved by our method under the lower budget are considerable. The plots on the top-right show power consumption and image errors during approximately 110 seconds of free camera navigation. Our method minimizes quality errors (bottom-right) while guaranteeing that power usage almost stays below the given budget. Please refer to the supplementary video for more details.*

TABLE 3: *Statistics of the six demo scenes, including the number of triangles, number of objects, and size on disk with textures.*

Scenes	#Triangles	#Objects	Scene Size	Platform
San Miguel	5.608 M	1,463	733 MB	Desktop
Bistro	3.858 M	1,285	2.73 GB	Desktop
Island	716.8 K	665	1.83 GB	Desktop
Subway	526.6 K	453	1.27 GB	Desktop
Valley	143.3 K	198	15.1 MB	Desktop & Mobile
Hall	229.4 K	22	338 MB	Desktop & Mobile

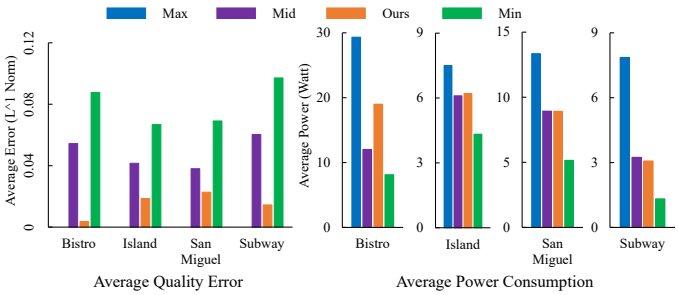

Fig. 9: *Average quality error (left) and power (right) per frame in our four desktop demos. The* Bistro *and the* Island *scenes are run on the Quadro P4000 graphics card, whereas the* San Miguel *and the* Subway *scenes are run on the RTX 2080Ti graphics card. The quality error of the maximum setting is zero, and the power budget used for the* San Miguel *scene in this figure is* 10 W*.*

and resolutions will not affect the performance and quality of network inferences since our networks do not take images as input. Given that frequent computations of quality errors can affect user experience and result in additional overheads, we set the Error Computation Interval to 3 seconds (90 frames). Here the main overhead on GPU is $1/90 \approx 1.1\%$ for rendering a reference frame and computing the error. The time window of temporal filtering is 2 seconds.

Given that inferences are computed at CPU, the costs of networks are negligible for GPU-bounded rendering. We did not observe significant improvement after implementing our networks on GPU but experience some impacts on the GPU's frequency and power consumption. Therefore, we only integrated the CPU implementation into our framework.

9 Results

To demonstrate the flexibility and effectiveness of our rendering framework, we performed a series of experiments with several scenes on different devices. In all experiments, we set a fixed frame rate of 30 FPS on the desktop platform. The frame rate for mobile devices is 10 FPS due to its limited computational power. The resolution is 1280×720 on both of the platforms. We tested six scenes with varying complexity. Their statistics are shown in Table 3. More details of these scenes are described in Section D of the supplementary document.

Our framework allows users to explore freely in the scene. For convenience, we designed a predefined camera path for each scene for measurement and comparison, and each path lasted at least 60 seconds. We used arbitrarily-chosen various power budgets to test our framework, with the maximum and the minimum rendering qualities as benchmarks. On the desktop platform, the reported power is subtracted from a baseline power measured on an empty scene because we focus on the net power of different rendering effects rather than the overall power of GPU. Figure 9 shows the average power consumption and quality error for the four scenes on the desktop platform. It can be seen that our framework can significantly reduce power consumption while keeping the error at a low level.

Generally, our two prediction networks trained from dummy scenes worked very well for the six test scenes with quite a different environment, lighting, geometry, etc. Figure 1 shows the *Bistro* scene running on the Quadro P4000 graphics card on a power budget $p_{\text{bgt}} = 25.5\ W$. The zoomed-in insets show that the results of our framework have minor differences compared with those of the maximum quality, whereas the minimum quality produces nu-

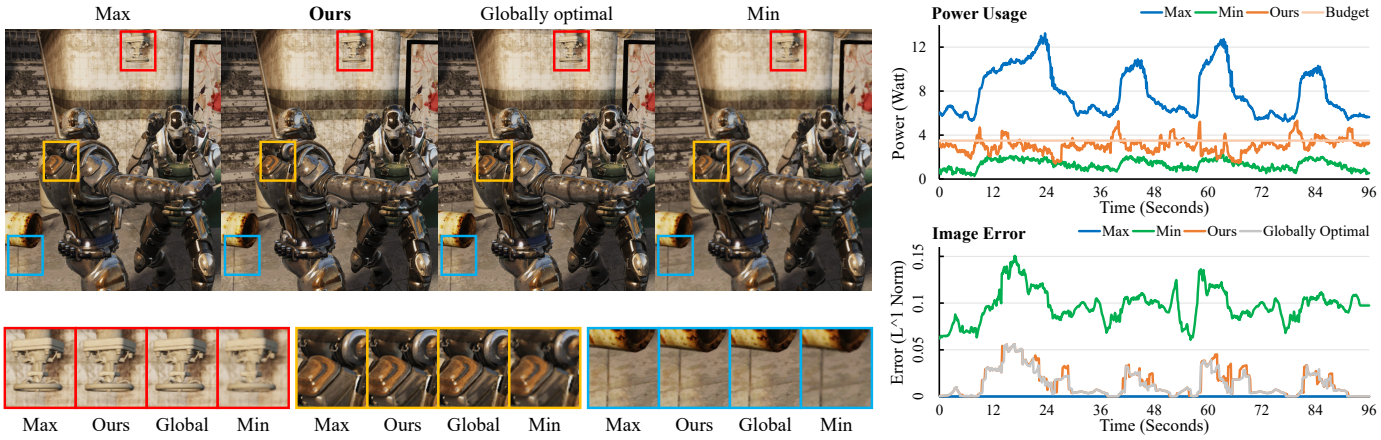

Fig. 10: *Real-time demo for the* Subway *scene, running with the RTX 2080Ti graphics card. We compare the maximum, globally optimal, and minimum settings against our power-budget rendering framework with a budget of* $3.5\ W$*. Plots on the right show power usage and quality errors during an approximately 96-second camera path. Our framework outputs images close to those produced with globally optimal settings (gray) while maintaining power usage below the budget on average. Please refer to the supplementary video for more details.*

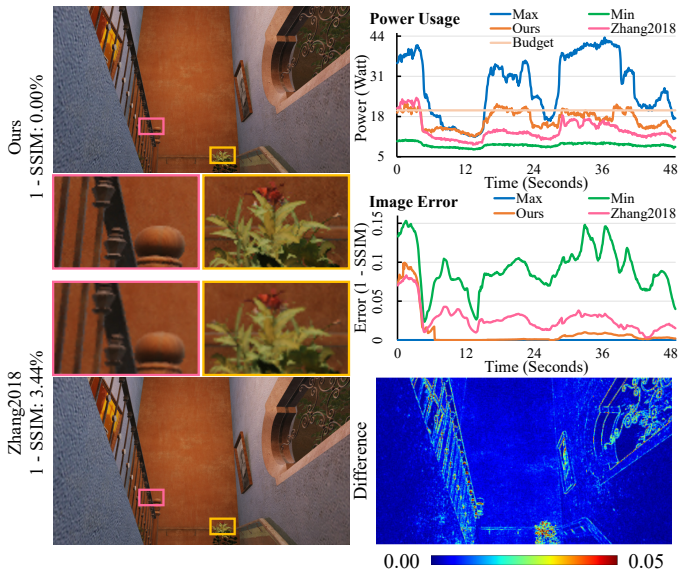

Fig. 11: *Comparison between our power-budget rendering framework and the previous online method [6] on a power budget of* $20\ W$*. This* San Miguel *scene is running on the P4000 graphics card. Plots on the right show power usage and quality errors during an approximately 48-second shot. The zoomed-in insets are shown on the middle left, and the difference between rendered images is shown on the right bottom.*

merous artifacts, such as blurred images and low-resolution shadows. The plots on the right show that the measured power of our method is mostly below the given budget, whereas the image error is significantly small. When the power usage of maximum quality is lower than the budget, our framework directly selects the best rendering setting to minimize quality error.

Figure 8 shows the *San Miguel* scene running on the RTX 2080Ti graphics card. During two explorations with different power budgets ($10\ W$ and $14\ W$), our framework generates optimal rendering settings continuously. At 48-60s, many leaves appeared in the camera viewport. However, a lot of fragments of these leaves were early discarded due to the alpha test. Our training dataset does not contain such an extreme case, so the selected setting went over the budget ($P_{\text{bgt}} = 10\ W$). Specializing the dataset could ameliorate this situation.

Figure 10 shows the *Subway* scene executed with the RTX 2080Ti graphics card on a power budget $p_{\text{bgt}} = 3.5\ W$. We compared the optimized settings based on our quality prediction model with the globally optimal settings. To find the globally optimal settings, we computed quality errors of all rendering settings whose predicted power was below the budget and then selected the one with minimum error as the globally optimal setting. In this case, the image errors produced by our framework is significantly close to the globally optimal settings, confirming the high usability of our quality prediction model.

We also compared our framework with manually set trade-offs. We set up a middle setting in which all of its quality levels are middle. As shown in Figure 9, our framework has similar energy consumption but fewer quality errors compared with the middle setting. On average, in the *Island* scene, our framework achieves a 55% improvement in visual quality at the expense of negligible power usage ($0.1\ W$). In the *San Miguel* and *Subway* scenes, we reach 44% and 76% improvement in quality, respectively, with nearly equal or less power consumption. These results fully demonstrate that our power-efficient framework can balance quality and power consumption automatically, which is a highly challenging work for manually adjusting the settings.

In addition, we made several comparisons with the previous online method [6]. Figure 11 shows one of the comparisons executed with the P4000 graphics card in the *San Miguel* scene. The power budget is set to $20\ W$. The linear prediction model of the previous method cannot be updated at each frame, which leads to wrong power predictions. Thus, the previous method fails to select the optimal rendering setting (i.e., directly select the maximum quality setting) at 8-15s, which reveals the weakness of linear regression and periodic fitting strategy. While the scene and view are changing, the linear regression based on the image quality from dozens of frames ago produces

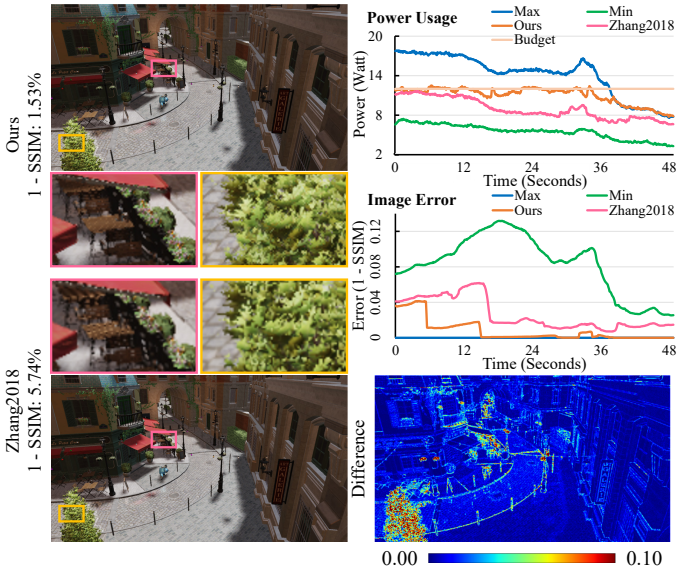

Fig. 12: *Comparison of the* Bistro *scene between our power-budget rendering framework and the previous method [6] on a budget of* 12 W*, running on the RTX 2080Ti graphics card.*

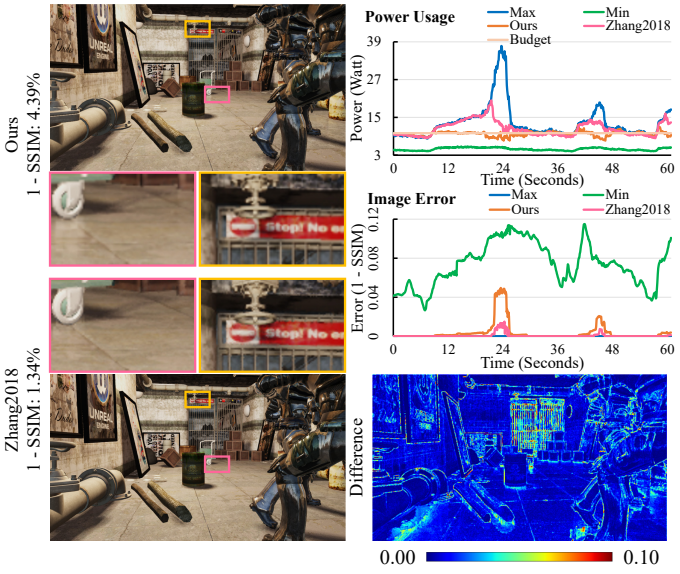

Fig. 13: *Comparison of the* Subway *scene between our power-budget rendering framework and the previous method [6] on a budget of* 10 W*, running with the Quadro P4000 graphics card.*

significant errors. Two visual examples are the staircase and the plant, where the wrongly chosen quality levels of resolution, metals, and anti-aliasing cause visual artifacts, such as the blurring and the loss of shine. In contrast, our learning-based per frame approach is robust on both power prediction and image error control, even if the scene and power usage dramatically change.

Figure 12 shows the *Bistro* scene running with the RTX 2080Ti graphics card on a power budget $p_{\text{bgt}} = 12\ W$. The previous method fails to predict power accurately, resulting in worse rendering settings selected around the 12th second. Figure 13 shows the *Subway* scene using the Quadro P4000 graphics card with a power budget $p_{\text{bgt}} = 10\ W$. Although the rendering settings selected by our method are of worse quality at 22-24s, the power consumption of the previous method goes over budget during this period.

Figure 14 shows a more representative scenario of current PC gaming requirements, wherein we set a fixed frame rate of 60 FPS. This *Island* scene was executed with the RTX 2080Ti graphics card on a power budget $p_{\text{bgt}} = 12\ W$. Like the previous comparison, Zhang et al. [6] go over the power budget at 36-44s. On the contrary, our approach almost always stays within budget while providing competitive rendering results.

In these comparisons, we still use the standardized L^1 norm metric to select the optimal rendering settings, but calculate the SSIM error of the selected settings to quantitatively compare quality errors.

More evaluations are shown in Section E of the supplementary document. Combined with all these results, we believe the proposed method can find the optimal rendering settings of various budgets on different GPUs using an identical network model with the same weights.

10 Discussion

Our power-budget rendering framework overcomes three limitations of the previous works [4], [6]. First, it no longer relies on view- and scene-related power measurements, which highly decreases the difficulty of collecting datasets. Besides, our power prediction model utilizes GPU's operating frequencies to provide high-accuracy predictions under various workloads. Finally, our framework achieves frame-by-frame power-budget rendering the first time in history, which self-tunes rendering parameters for each frame in any new scene or view. We show the optimization results of six typical scenarios with high-accuracy predictions and selections when the workload changes significantly. These results demonstrate the effectiveness and flexibility of our framework. Our rendering system does not introduce any frame rate loss, as shown in the supplementary video.

Setting the power budget to an absolute value is useful in many cases. For example, when running video games on a smartphone, one can estimate the minimum duration of the battery by setting a reasonable power budget. Additionally, an absolute budget also provides the rendering system an opportunity to select the best quality setting.

In some cases, the measured power consumption may slightly deviate from the power budget because of the accuracy of the generic power prediction model. Given that the current power measurement method cannot guarantee a sufficiently small variance, the generated dataset contains errors. Using such a dataset to train a highly flexible network will slightly reduce the prediction accuracy. However, our power prediction model can specialize in customized scenes or specific hardware platforms for increased accuracy.

Typically, fixing the Error Computation Interval to an appropriate value can satisfy various situations. This interval can also be adjusted automatically at runtime. For example, the framework can perform an error computation process only when the framework detects that the percentage occupied areas of the current frame are highly different from the previous frame.

Some parts of our framework can still be studied further. Our power prediction network only considers the three pipeline stages of draw calls, vertices, and fragments. However, the latest rendering pipeline, such as DirectX Raytracing [50], has included some new rendering stages

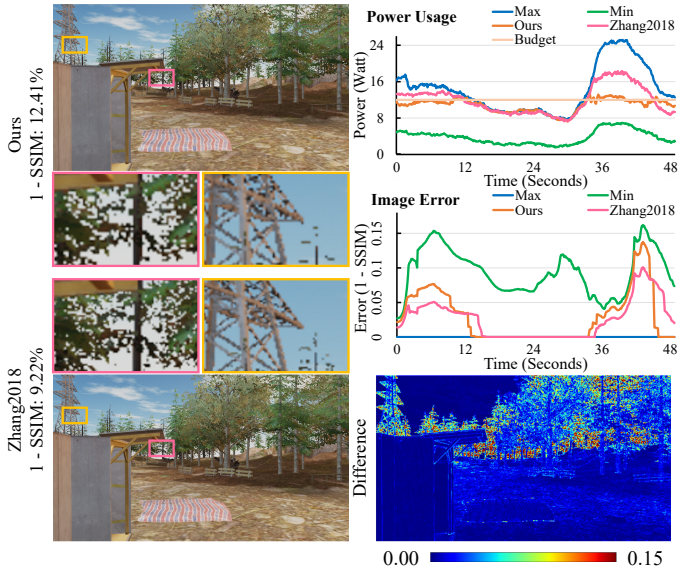

Fig. 14: *Comparison tested at 60 FPS for the* Island *scene between our power-budget rendering framework and the previous online method [6] on a budget of* 12 *W, running with the RTX 2080Ti graphics card.*

such as ray tracing. Extending our framework to a new pipeline is an interesting research topic for future work. In addition, further reducing the costs of quality error prediction remains a topic to be explored.

Our system can be integrated into the rendering engine like an add-on. For various applications using the same engine and rendering settings, our networks can generalize across them without any retraining. However, a retraining step is needed for applications using different engines or an identical engine with different rendering settings.

In sum, our real-time power-budget rendering framework satisfies the following requirements of an ideal power-saving framework [4]: 1) it finds the optimal trade-off between energy and quality; 2) users can specify the target energy consumption to extend the battery life; 3) it is real-time and transparent to the users; and 4) it generalizes across different platforms and scenes. We show the results of six scenes on four devices of two platforms. We also demonstrate that our framework outperforms hand-setting configurations. Fewer overheads and better flexibility of our system allow the integration with existing rendering engines at negligible costs.

Acknowledgment

We would like to thank all reviewers and editors for their insightful comments. This research was partially funded by NSFC (No. 61872319), Zhejiang Provincial NSFC (No. LR18F020002), and National Key R&D Program of China (No. 2017YFB1002605).

References

[1] K. W. Nixon, X. Chen, H. Zhou, Y. Liu, and Y. Chen, "Mobile GPU power consumption reduction via dynamic resolution and frame rate scaling," in *6th Workshop on Power-Aware Computing and Systems (HotPower 14)*. Broomfield, CO: USENIX Association, 2014.

[2] E. Stavrakis, M. Polychronis, N. Pelekanos, A. Artusi, P. Hadjichristodoulou, and Y. Chrysanthou, "Toward energy-aware balancing of mobile graphics," in *IS&T/SPIE Electronic Imaging, International Society for Optics and Photonics*, vol. 9411, 2015, pp. 94 110D–10.

[3] S. He, Y. Liu, and H. Zhou, "Optimizing smartphone power consumption through dynamic resolution scaling," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '15. New York, NY, USA: ACM, 2015, pp. 27–39.

[4] R. Wang, B. Yu, J. Marco, T. Hu, D. Gutierrez, and H. Bao, "Real-time rendering on a power budget," *ACM Trans. Graph.*, vol. 35, no. 4, pp. 111:1–111:11, Jul. 2016.

[5] C. Hwang, S. Pushp, C. Koh, J. Yoon, Y. Liu, S. Choi, and J. Song, "Raven: Perception-aware optimization of power consumption for mobile games," in *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '17. New York, NY, USA: ACM, 2017, pp. 422–434.

[6] Y. Zhang, M. Ortin, V. Arellano, R. Wang, D. Gutierrez, and H. Bao, "On-the-fly power-aware rendering," *Computer Graphics Forum*, vol. 37, no. 4, pp. 155–166, 2018.

[7] T. Davidovič, J. Křivánek, M. Hašan, and P. Slusallek, "Progressive light transport simulation on the gpu: Survey and improvements," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 3, pp. 1–19, 2014.

[8] M. Gharbi, T.-M. Li, M. Aittala, J. Lehtinen, and F. Durand, "Sample-based monte carlo denoising using a kernel-splatting network," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 4, pp. 1–12, 2019.

[9] Y. Huo, S. Jin, T. Liu, W. Hua, R. Wang, and H. Bao, "Spherical gaussian-based lightcuts for glossy interreflections," in *Computer Graphics Forum*. Wiley Online Library, 2020.

[10] Y. Huo, R. Wang, T. Hu, W. Hua, and H. Bao, "Adaptive matrix column sampling and completion for rendering participating media," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 6, pp. 1–11, 2016.

[11] B. Moon, J. A. Iglesias-Guitian, S.-E. Yoon, and K. Mitchell, "Adaptive rendering with linear predictions," *ACM Transactions on Graphics (TOG)*, vol. 34, no. 4, pp. 1–11, 2015.

[12] T. Vogels, F. Rousselle, B. McWilliams, G. Röthlin, A. Harvill, D. Adler, M. Meyer, and J. Novák, "Denoising with kernel prediction and asymmetric loss functions," *ACM Transactions on Graphics (TOG)*, vol. 37, no. 4, pp. 1–15, 2018.

[13] Y. Huo, R. Wang, R. Zheng, H. Xu, H. Bao, and S.-E. Yoon, "Adaptive incident radiance field sampling and reconstruction using deep reinforcement learning," *ACM Transactions on Graphics (TOG)*, vol. 39, no. 1, pp. 1–17, 2020.

[14] D. Gao, X. Li, Y. Dong, P. Peers, K. Xu, and X. Tong, "Deep inverse rendering for high-resolution svbrdf estimation from an arbitrary number of images," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 4, p. 134, 2019.

[15] R. Wang, Y. Huo, Y. Yuan, K. Zhou, W. Hua, and H. Bao, "Gpu-based out-of-core many-lights rendering," *ACM Transactions on Graphics (TOG)*, vol. 32, no. 6, pp. 1–10, 2013.

[16] Y. Huo, R. Wang, S. Jin, X. Liu, and H. Bao, "A matrix sampling-and-recovery approach for many-lights rendering," *ACM Transactions on Graphics (TOG)*, vol. 34, no. 6, pp. 1–12, 2015.

[17] K. Xu, W.-L. Sun, Z. Dong, D.-Y. Zhao, R.-D. Wu, and S.-M. Hu, "Anisotropic spherical gaussians," *ACM Transactions on Graphics (TOG)*, vol. 32, no. 6, pp. 1–11, 2013.

[18] Y. He, Y. Gu, and K. Fatahalian, "Extending the graphics pipeline with adaptive, multi-rate shading," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 4, pp. 1–12, 2014.

[19] P. Lecocq, A. Dufay, G. Sourimant, and J. Marvie, "Analytic approximations for real-time specular area lighting," *IEEE Transactions on Visualization & Computer Graphics*, vol. 23, no. 05, pp. 1428–1441, may 2017.

[20] M. McGuire and M. Mara, "Phenomenological transparency," *IEEE Transactions on Visualization & Computer Graphics*, vol. 23, no. 05, pp. 1465–1478, may 2017.

[21] E. Heitz, J. Dupuy, S. Hill, and D. Neubelt, "Real-time polygonal-light shading with linearly transformed cosines," *ACM Trans. Graph.*, vol. 35, no. 4, Jul. 2016.

[22] J. Dupuy, E. Heitz, and L. Belcour, "A spherical cap preserving parameterization for spherical distributions," *ACM Trans. Graph.*, vol. 36, no. 4, Jul. 2017.

[23] L. Belcour and P. Barla, "A practical extension to microfacet theory for the modeling of varying iridescence," *ACM Trans. Graph.*, vol. 36, no. 4, Jul. 2017.

[24] A. Silvennoinen and J. Lehtinen, "Real-time global illumination by precomputed local reconstruction from sparse radiance probes," *ACM Trans. Graph.*, vol. 36, no. 6, Nov. 2017.
[25] M. Dong and L. Zhong, "Power modeling and optimization for oled displays," *IEEE Transactions on Mobile Computing*, vol. 11, no. 9, pp. 1587–1599, Sep. 2012.
[26] W. Chen, W. Chen, H. Chen, Z. Zhang, and H. Qu, "An energy-saving color scheme for direct volume rendering," *Computers & Graphics*, vol. 54, pp. 57 – 64, 2016.
[27] M. Weiser, B. Welch, A. Demers, and S. Shenker, *Scheduling for Reduced CPU Energy*. Boston, MA: Springer US, 1996, pp. 449–471.
[28] S. Kim, T. Harada, and Y. J. Kim, "Energy-efficient global illumination algorithms for mobile devices using dynamic voltage and frequency scaling," *Computers & Graphics*, vol. 70, pp. 198–205, 2018.
[29] J. Pool, A. Lastra, and M. Singh, "Precision selection for energy-efficient pixel shaders," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, ser. HPG '11. New York, NY, USA: ACM, 2011, pp. 159–168.
[30] J. Ragan-Kelley, J. Lehtinen, J. Chen, M. Doggett, and F. Durand, "Decoupled sampling for graphics pipelines," *ACM Trans. Graph.*, vol. 30, no. 3, pp. 17:1–17:17, May 2011.
[31] P. Clarberg, R. Toth, J. Hasselgren, J. Nilsson, and T. Akenine-Möller, "Amfs: Adaptive multi-frequency shading for future graphics processors," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 141:1–141:12, Jul. 2014.
[32] PowerVR, "A master class in graphics technology and optimization," 2012. http://imgtec.eetrend.com/sites/imgtec.eetrend.com/files/download/201402/1458-2110-1385011428.pdf
[33] P. Mathias, "Arm mali gpu midgard architecture," 2016. http://imgtec.eetrend.com/sites/imgtec.eetrend.com/files/download/201402/1458-2110-1385011428.pdf
[34] J. M. Vatjus-Anttila, T. Koskela, and S. Hickey, "Power consumption model of a mobile GPU based on rendering complexity," in *2013 Seventh International Conference on Next Generation Mobile Apps, Services and Technologies*, Sept 2013, pp. 210–215.
[35] C. Huang, Y. Chung, P. Huang, and S. Tsao, "High-level energy consumption model of embedded graphic processors," in *2015 IEEE International Conference on Digital Signal Processing (DSP)*, July 2015, pp. 105–109.
[36] B. Xu, J. Zhang, R. Wang, K. Xu, Y.-L. Yang, C. Li, and R. Tang, "Adversarial monte carlo denoising with conditioned auxiliary feature modulation," *ACM Trans. Graph.*, vol. 38, no. 6, pp. 224:1–224:12, Nov. 2019.
[37] C. R. A. Chaitanya, A. S. Kaplanyan, C. Schied, M. Salvi, A. Lefohn, D. Nowrouzezahrai, and T. Aila, "Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder," *ACM Trans. Graph.*, vol. 36, no. 4, pp. 98:1–98:12, Jul. 2017.
[38] L. Yang, Z. Shi, Y. Zheng, and K. Zhou, "Dynamic hair modeling from monocular videos using deep neural networks," *ACM Trans. Graph.*, vol. 38, no. 6, pp. 235:1–235:12, Nov. 2019.
[39] D. Vicini, V. Koltun, and W. Jakob, "A learned shape-adaptive subsurface scattering model," *ACM Trans. Graph.*, vol. 38, no. 4, pp. 127:1–127:15, Jul. 2019.
[40] L.-Q. Yan, W. Sun, H. W. Jensen, and R. Ramamoorthi, "A bssrdf model for efficient rendering of fur with global illumination," *ACM Trans. Graph.*, vol. 36, no. 6, pp. 208:1–208:13, Nov. 2017.
[41] K. Zsolnai-Fehér, P. Wonka, and M. Wimmer, "Gaussian material synthesis," *ACM Trans. Graph.*, vol. 37, no. 4, pp. 76:1–76:14, Jul. 2018.
[42] O. Nalbach, E. Arabadzhiyska, D. Mehta, H.-P. Seidel, and T. Ritschel, "Deep shading: Convolutional neural networks for screen space shading," *Computer Graphics Forum*, vol. 36, no. 4, pp. 65–78, 2017.
[43] P. Ren, J. Wang, M. Gong, S. Lin, X. Tong, and B. Guo, "Global illumination with radiance regression functions," *ACM Trans. Graph.*, vol. 32, no. 4, pp. 130:1–130:12, Jul. 2013.
[44] E. Le Sueur and G. Heiser, "Dynamic voltage and frequency scaling: The laws of diminishing returns," in *Proceedings of the 2010 international conference on Power aware computing and systems*, 2010, pp. 1–8.
[45] C. Dugas, Y. Bengio, F. Bélisle, C. Nadeau, and R. Garcia, "Incorporating second-order functional knowledge for better option pricing," in *Advances in neural information processing systems*, 2001, pp. 472–478.
[46] NVML, "Nvidia management library," 2019. https://developer.nvidia.com/nvidia-management-library-nvml
[47] T. Stachowiak, "Stochastic screen-space reflections," in *ACM SIGGRAPH 2015 Courses Advances in Real-Time Rendering in Games*, 2015.
[48] J. Jimenez, D. Gutierrez, J. Yang, A. Reshetov, P. Demoreuille, T. Berghoff, C. Perthuis, H. Yu, M. McGuire, T. Lottes, H. Malan, E. Persson, D. Andreev, and T. Sousa, "Filtering approaches for real-time anti-aliasing," in *ACM SIGGRAPH 2011 Courses*, ser. SIGGRAPH '11. New York, NY, USA: ACM, 2011, pp. 6:1–6:329.
[49] J. Jimenez, J. I. Echevarria, T. Sousa, and D. Gutierrez, "SMAA: Enhanced morphological antialiasing," *Computer Graphics Forum (Proc. EUROGRAPHICS 2012)*, vol. 31, no. 2, 2012.
[50] C. Wyman, S. Hargreaves, P. Shirley, and C. Barré-Brisebois, "Introduction to directx raytracing," in *ACM SIGGRAPH 2018 Courses*, ser. SIGGRAPH '18. New York, NY, USA: ACM, 2018, pp. 9:1–9:1.

Yunjin Zhang Yunjin Zhang received the bachelor's degree in Software Engineering from University of Electronic Science and Technology of China, in 2016. He is working toward a Ph.D. degree in the State Key Laboratory of CAD&CG, Zhejiang University. His interests are mainly in the optimization framework for real-time rendering, especially power-budget rendering, VR/AR optimization, and cloud-client co-computing rendering.

Rui Wang Rui Wang is a professor at the State Key Laboratory of CAD&CG, Zhejiang University. He received a bachelor's degree in Computer Science and a Ph.D. degree in Mathematics from Zhejiang University. His research interests are mainly in real-time rendering, realistic rendering, GPU-based computation, and 3D display techniques. Now, he is leading a group working on the next-generation real-time rendering engine and techniques.

Yuchi Huo Yuchi Huo is graduated from Zhejiang University and working at SGVR Lab. at KAIST. His research interests are in rendering, deep learning, image processing, and computational optics.

Wei Hua Wei Hua is an associate professor with the State Key Laboratory of CAD&CG, Zhejiang University. He got a Ph.D. in Applied Mathematics from Zhejiang University in 2002. He joined the CAD&CG State Key Lab in 2002. His main interests include real-time simulation and rendering, virtual reality, and software engineering.

Hujun Bao Hujun Bao is a professor with the State Key Laboratory of CAD&CG and the College of Computer Science and Technology, Zhejiang University. He leads the 3D graphics computing group in the lab, which mainly makes researches on geometry computing, 3D visual computing, real-time rendering, and their applications. His research goal is to investigate the fundamental theories and algorithms to achieve good visual perception for interactive digital environments, and develop related systems.