Real-time Rendering on a Power Budget

Rui Wang[1]* Bowen Yu[1] Julio Marco[2] Tianlei Hu[1] Diego Gutierrez[2,3] Hujun Bao[1]*

[1] State Key Lab of CAD&CG, Zhejiang University [2] Universidad de Zaragoza [3] I3A Institute

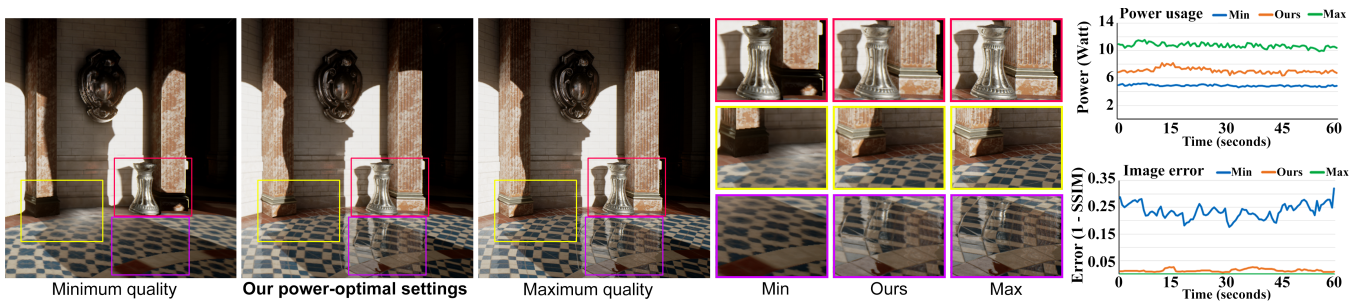


Figure 1: *We propose a novel framework that dynamically yields the optimal rendering settings to minimize power consumption while maximizing visual quality, in real-time. The figure shows results for the* Sun Temple *scene, where our Power-Optimal settings yield images of similar quality as Maximum Quality, but saving up to* 34% *in power consumption. The charts on the right show power consumption and image quality (measured with the perceptually-based SSIM metric), respectively.*

Abstract

With recent advances on mobile computing, power consumption has become a significant limiting constraint for many graphics applications. As a result, rendering on a power budget arises as an emerging demand. In this paper, we present a real-time, power-optimal rendering framework to address this problem, by finding the *optimal* rendering settings that minimize power consumption while maximizing visual quality. We first introduce a novel power-error, multi-objective cost space, and formally formulate power saving as an optimization problem. Then, we develop a two-step algorithm to efficiently explore the vast power-error space and leverage optimal Pareto frontiers at runtime. Finally, we show that our rendering framework can be generalized across different platforms, desktop PC or mobile device, by demonstrating its performance on our own OpenGL rendering framework, as well as the commercially available Unreal Engine.

Keywords: power-optimal rendering, rendering system

Concepts: •**Computing methodologies** → **Rendering;**

1 Introduction

The increasing incorporation of GPUs on mobile, battery-powered devices during the last years has led to the emergence of many real-time rendering applications. These applications and the required computations, however, demand a high energy consumption. This has a significant impact on battery life, which becomes a limiting constraint for mobile devices. As a consequence, lowering the energy requirements on rendering applications has been recently identified as one of the next challenges in computer graphics [Peddie 2013]. However, a generalized methodology does not exist yet, and its possibilities remain largely unexplored.

Among the explored strategies to reduce energy consumption for graphics applications running on battery-powered devices, reducing the number of computations on the rendering pipeline has proved to be an effective solution (e.g., [Woo et al. 2002; Pool 2012; Johnsson 2014; Arnau et al. 2014; Stavrakis et al. 2015]). However, most existing solutions are based on ad-hoc decisions, tailored to a particular application. While previous works aiming to reduce computations in real-time rendering have relied on multi-objective cost functions defined by visual error, rendering time, or memory consumption [Pellacini 2005; Sitthi-amorn et al. 2011; Wang et al. 2015; He et al. 2015], we introduce a new cost model based on visual quality and *power usage*.

An ideal power-saving framework should have the following characteristics: 1) It guarantees an *optimal* tradeoff between the quality of the results and the target energy footprint; 2) The user can *adjust* both a target quality or a target energy consumption to prolong battery life; 3) It is *real-time*, and transparent to the user; 4) It *generalizes* across platforms and applications.

Finding the optimal settings from the usually huge set of rendering parameters available in graphics applications is a very challenging task, which requires an intelligent exploration of the large power-error space. This is further complicated by the desired real-time and multi-platform requirements. In this paper, we address these challenges and present a real-time, power-optimal rendering framework that automatically finds the optimal tradeoffs between power consumption and image quality, and adapts the required rendering settings dynamically at run-time. We demonstrate how our adaptive exploration of the energy footprint of a rendering application can be leveraged to reduce power usage while preserving quality on the results. In particular our contributions are:

- We formally formulate the power vs. error tradeoff as an op-

*Corresponding authors: Rui Wang (rwang@cad.zju.edu.cn), Hujun Bao (bao@cad.zju.edu.cn)

SIGGRAPH '16 Technical Paper, July 24 - 28, 2016, Anaheim, CA,
ISBN: 978-1-4503-4279-7/16/07
DOI: http://dx.doi.org/10.1145/2897824.2925889

timization problem, and present a multi-objective cost model defined in a novel power-error space.

- Based on this model, we present a new two-stage rendering framework that efficiently explores the power-error space, and adaptively reduces rendering costs at run-time.
- We demonstrate the flexibility and effectiveness of our framework using both a custom-built, OpenGL rendering system, tested on a smartphone and a desktop PC, and the commercial renderer integrated in the Unreal Engine, running on a desktop PC.

2 Related Work

Energy-aware devices and algorithms are becoming a prolific research topic, with recent examples in fields like data management [Beloglazov et al. 2012], systems design [Kyung and Yoo 2014], cloud photo enhancement [Gharbi et al. 2015], or display technology [Masia et al. 2013], to name a few. This last one is maybe the field where energy consumption has been more thoroughly researched, while power-efficient rendering algorithms are increasingly drawing attention, largely motivated by the widespread adoption of mobile devices.

Power Saving for Displays In the last decade or so, many existing works have focused on reducing energy consumption in displays [Moshnyaga and Morikawa 2005; Shearer 2007]. For back-lit LCD displays, most of the light is converted to heat, a problem that is aggravated for HDR displays [Masia et al. 2013]. Dimming is the most common energy-saving strategy, e.g., simply by reducing the intensity of the background light [Narra and Zinger 2004], darkening inactive regions [Iyer et al. 2003], or by concurrent brightness and contrast scaling [Cheng and Pedram 2004]. More modern OLED displays allow energy control at individual pixel level [Forrest 2003; Dong and Zhong 2012], which enables more sophisticated strategies like saliency-based dimming [Chen et al. 2014]. Energy-efficient color schemes have been proposed, for instance as a set of distinguishable iso-lightness colors guided by perceptual principles [Chuang et al. 2009], or by finding a suitable color palette by means of energy minimization [Dong et al. 2009]. Chen et al. [2016] present an optimization approach for volume rendering, optimizing color sets in object space instead of image space. Vallerio and colleagues [2006] and Ranganathan et al. [2006] explore energy efficiency for displays in the context of designing user interfaces.

Power Saving for GPUs With the establishment of GPUs and mobile devices, several specific pipeline designs and hardware implementations have been developed to optimize resources and reduce power usage during rendering (see for instance [Woo et al. 2002; Pool 2012; Stavrakis et al. 2015]). Möller and Ström [2008] presented a survey about GPU design, where power consumption plays a key role, while the recent thesis by Johnsson [2014] offers for a more detailed discussion of hardware-related aspects concerning power usage. Arnau et al. [2014] reduce mobile GPUs energy consumption by removing redundancy of fragment shaders operations at hardware level. Instead, we present a purely software-driven power optimization strategy, agnostic to the underlying hardware being used.

Tile-based deferred rendering (TBDR) [PowerVR 2012] identifies the portions of the scenes that can be ignored in the very early stages of rendering, therefore saving GPU computation and power consumption. Johnsson et al. [2012] compared the power efficiency of three rendering and three shadow algorithms on different GPUs, although they do not provide new energy-efficient algorithms. Recently, Cohade and Santos [2015] presented their efforts on optimizing the power usage in the *Lego Minifigures* game, and Mavridis and Papaioannou [2015] reported energy savings on GPU-implementation of coarse shading techniques [Vaidyanathan et al. 2014]. Different from these works, our approach makes use of actual energy consumption and error measurements, to drive a real-time power-optimal rendering system.

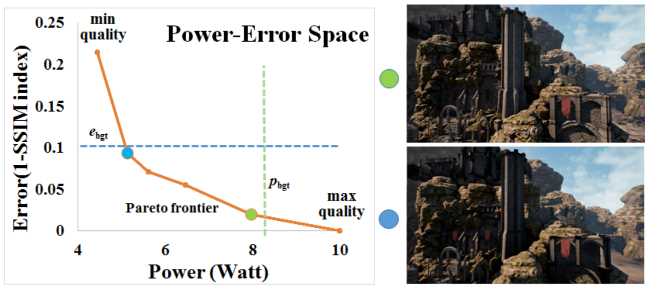


Figure 2: *Illustration of the power-optimal rendering. With Pareto-optimal rendering settings, it is possible to obtain the optimal tradeoffs between power and visual error. Two rendering settings, marked in blue and green, are the optimal rendering settings with respect to the error budget e_{bgt} and the power budget p_{bgt}. One achieves the minimum power under the error budget, and the other obtain the minimum visual error under the power budget.*

Complementary to power optimization, other rendering resources such as memory bandwidth or computation time have been the focus of different optimization schemes [Wang et al. 2015; He et al. 2015], aiming for a good tradeoff between image quality and rendering budgets. Complementary to these works, we aim to find an optimal compromise between image quality and a new challenging and constraining budget: energy consumption.

3 Problem Definition

In our context, it is useful to think about the rendering process as a function f that performs multiple rendering passes[1], and returns a color image. Each rendering function takes as input, on the one hand, the rendering settings s, defining the visual effects (shadow mapping, screen-space ambient occlusion, etc) and the specific parameters used for each one (such as map resolutions or kernel sizes); and on the other hand, the camera parameters c (position and view). It is clear that different rendering settings yield images with different quality for a given camera.

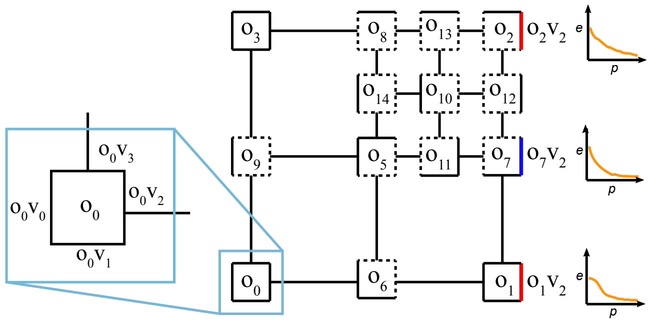
Let s_{best} denote the rendering settings that generate the best quality image. We can define the quality error e of any other image produced by different rendering settings as

$$e(c, s) = \iint_{xy} \| \, f(c, s_{\text{best}}) - f(c, s) \, \| \, \mathrm{d}xy \qquad (1)$$

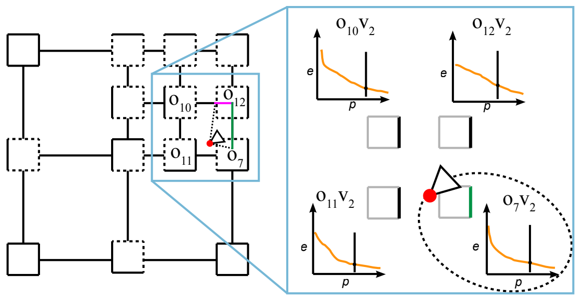
where x, y define the pixel domain of the image, and $\| \cdot \|$ indicates the chosen norm.

Rendering with different functions $f(c, s)$ also has an impact on power usage. We can denote the power consumed during rendering of one frame as $p(c, s)$. In general, higher-quality images require more power, while rendering a minimum quality image can save

[1]Generalizing the rendering process as a function f allows us to include both forward and deferred rendering frameworks.



(a) *Adaptive measurement in an octree*



(b) *Runtime power optimal rendering*

Figure 3: *Overview of our power-optimal rendering process. For the sake of simplicity, we illustrate a 2D example using a quad-tree.* ***(a) Adaptive subdivision:*** *The initial node has four corners $o_{0..3}$, where each corner o_i defines four axis-aligned views $v_0..v_3$ (light blue zoomed-in node). Cameras $c = (o_i, v_j)$ are placed at every position-view pair, where we compute their Pareto frontiers. For each pair of adjacent camera samples looking at the same view —e.g. o_2, v_2 and o_1, v_2, highlighted in red— if either the error or power difference between their Pareto frontiers is larger than a threshold, we subdivide the corresponding node and compute the Pareto frontier of the new camera sample (o_7, v_2), highlighted in blue. Otherwise, the Pareto frontiers of the new cameras are inherited (dashed lines). This process is repeated for each node until a certain depth level or given error and power difference thresholds.* ***(b) Rendering settings at run-time:*** *Given a camera position and its node in our structure, we select the closest camera sample and corresponding view (o_7, v_2) (green). The optimal rendering settings are then obtained from its Pareto frontier.*

over 50% of the power compared to the maximum quality (see Table 3). It is therefore possible to find suitable tradeoffs between quality and power usage, to either obtain the best rendering quality under a given power budget, or to ensure a minimal power consumption given a desired rendering quality. We call this *power-optimal rendering*. The optimization for a given power budget p_{bgt} can be formulated as

$$s = \arg\min_s e(c, s) \qquad \text{subject to} \qquad p(c, s) < p_{\text{bgt}}, \qquad (2)$$

whereas given a target quality defined by the error budget e_{bgt}, the optimization becomes

$$s = \arg\min_s p(c, s) \qquad \text{subject to} \qquad e(c, s) < e_{\text{bgt}}. \qquad (3)$$

4 Power-Optimal Rendering

We formulate our power-optimal rendering approach as a multi-objective optimization in a visual quality and power usage space. In this section we introduce our multi-objective cost model and the basic idea to solve the power-optimal rendering problem.

4.1 Multi-objective Cost Model

Different from other works [Pellacini 2005; Sitthi-amorn et al. 2011; Wang et al. 2015; He et al. 2015], our novel multi-objective cost model is based on visual quality and *power usage*. To optimize the rendering settings s of a given camera c, we first introduce a partial order to compare two different rendering settings s_i and s_j, and say that s_i is preferred over s_j (written as $s_i \prec s_j$) if either $e(c, s_i) < e(c, s_j) \wedge p(c, s_i) \leq p(c, s_j)$ or $e(c, s_i) \leq e(c, s_j) \wedge p(c, s_i) < p(c, s_j)$. That is, one rendering setting is preferred over another if it improves in quality or power usage, and is at least as good in the other.

Using our partial order, the Pareto frontier of all rendering settings $P(U) = \{u \in U : \forall u' \in U, u \not\prec u'\}$, can be regarded as the curve defining all power-optimal rendering settings in our two-dimensional cost space defined by (e, p). That is, the rendering settings in the Pareto frontier are preferred over other settings.

Working in the domain of the Pareto frontier has one key advantage: given a power budget or an error budget, finding the optimal rendering settings is reduced to a 1D search on the Pareto curve. As we will see, this dimensionality reduction is a crucial aspect which will allows us to select optimal rendering settings at run-time. Figure 2 shows an example Pareto frontier, from which two optimal rendering settings have been selected (given a power budget and an error budget respectively). The resulting images are shown on the right.

4.2 Adaptive Partition of The Camera View-Space

By optimizing our multi-objective cost model, we have the solution of Equations 2 and 3 for one particular camera. However, given the high dimensionality of the camera view-space (composed of all possible camera positions and views) it would be impractical to carry out all Pareto-optimal optimization at run-time. Therefore, we introduce an adaptive partition of the camera view-space to store precomputed, optimized Pareto frontiers at given positions and views, which will later enable real-time optimal rendering. Such an adaptive partition is based on the observation that at some regions in the camera view-space, the rendering settings on the Pareto frontiers are quite different, but at other regions they are similar.

In particular, we use an octree structure, where one corner of octree node defines a camera position, and defines a discrete set of six views at each position, forming a view cube. Each position-view pair (o_i, v_j) thus describes one camera sample, $c = (o_i, v_j)$. For the sake of clarity, a simplified 2D version is shown in Figure 3a. At each position-view, we compute the Pareto frontier representing the optimal tradeoffs between power usage and rendering quality. The differences between these frontiers for adjacent (o_i, v_j) pairs will guide the adaptive partition of the space. In practice, we found that adaptive spatial subdivision along with six view orientations maintains a good tradeoff between structure complexity, temporal smoothness, and computational cost. A complete description of this process is described in Section 5.

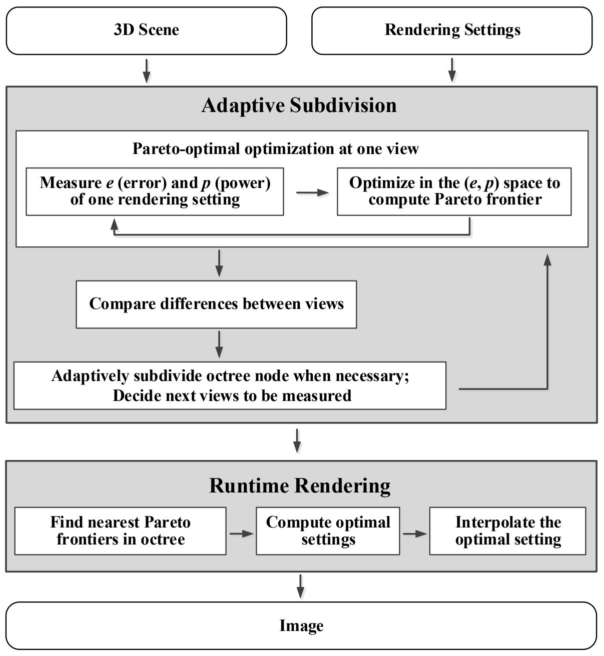

3D Scene
Rendering Settings
Adaptive Subdivision
Pareto-optimal optimization at one view
Measure *e* (error) and *p* (power) of one rendering setting
Optimize in the (*e*, *p*) space to compute Pareto frontier
Compare differences between views
Adaptively subdivide octree node when necessary; Decide next views to be measured
Runtime Rendering
Find nearest Pareto frontiers in octree
Compute optimal settings
Interpolate the optimal setting
Image

Figure 4: *Our algorithm is split up into two main stages: the adaptive measurement stage (described in Section 5) followed by the runtime rendering stage (described in Section 6).*

4.3 Algorithm Overview

Figure 4 shows an overview of our algorithm, based on our multi-objective cost model, and our adaptive partition of the camera view-space. Our input is a 3D scene and a set of rendering effects and parameters (see Table 1 for the set used in our implementation). The entire algorithm is split into two main stages: the adaptive subdivision stage and the run-time rendering. As a preprocess, from our initial octree node, we measure the error in visual quality e and the power usage p for each camera c, exploring the space of all possible rendering settings s through Genetic Programming; this yields the Pareto frontier for such camera. We then compare the Pareto frontiers of each pair of adjacent cameras sharing the same view direction, and subdivide the octree if the difference is too large. We iteratively repeat this process until no more subdivisions are needed. At run-time, novel views can be rendered under the given quality or power budget by interpolating the optimal rendering settings at the nearest sample positions and views during user exploration of the scene. The following sections offers details on each of the main steps.

5 Adaptive Subdivision

As a preprocess, the adaptive subdivision partitions the camera view-space and stores Pareto-frontiers at sampled camera positions and views. It mainly takes following three steps.

5.1 Pareto-Optimal Optimization at One Camera

5.1.1 Error and Power Measurement

Given a camera c, we first render its reference image using the maximum quality settings. This image will then be used to compare the output of all other rendering settings, according to Equation 1. Instead of relying on pixel-wise error metrics such as the L^2 norm, we use the Structural Similarity Index (SSIM) [Wang et al. 2004]

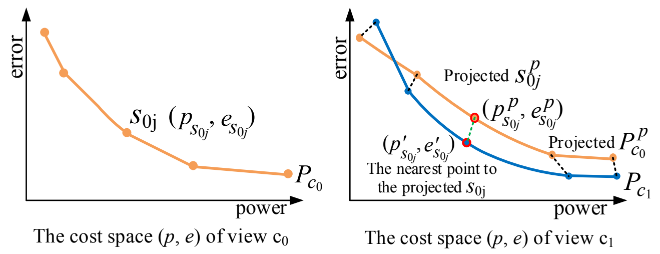

error
power
s_{0j} $(p_{s_{0j}}, e_{s_{0j}})$
P_{c_0}
The cost space (p, e) of view c_0
Projected s_{0j}^{p}
$(p_{s_{0j}}^{p}, e_{s_{0j}}^{p})$
$(p'_{s_{0j}}, e'_{s_{0j}})$
The nearest point to the projected s_{0j}
Projected $P_{c_0}^{p}$
P_{c_1}
The cost space (p, e) of view c_1

Figure 5: *Illustration of the distance from the Pareto frontiers P_{c_0} (left, orange) to P_{c_1} (right, blue). One distance between the error and power $(e_{s_{0j}}, p_{s_{0j}})$ of projected points s_{0j} to nearest point $(e'_{s_{0j}}, p'_{s_{0j}})$ is visualized in green dash line (right). The total distance is averaged by all point-wise distance (black dashed lines) to the projected $P_{c_0}^{p}$ (right, orange).*

to measure the similarity and use one minus the similarity to obtain the error, i.e $e = 1 - \text{SSIM}$, which yields results that better predict human perception.

To measure power usage, we use two different approaches, depending on the target platform (desktop PC or mobile device). For the desktop PC, we use specific APIs provided by GPU vendors to access the hardware's internal power readings, and read back power consumption. For the mobile device we measure it directly instead, since we did not find any reliable APIs to measure power; we remove the battery and use an external metered power source to read power usage. More details are provided in the supplementary document.

5.1.2 Exploring Potentially Optimal Settings

For each camera c, the space of all possible rendering settings s is large. For an efficient exploration of such space, we rely on Genetic Programming (GP), inspired by recent works on shader simplification [Sitthi-amorn et al. 2011; Wang et al. 2015]. Given its speed, we adapt the algorithm proposed by Deb et al. [2002], which fits our multi-objective optimization in error and power space well.

First, we randomly combine parameters of rendering settings to generate the initial population. During partition, after every subdivision of the octree, children nodes are initialized by inheriting the optimal rendering settings of the parent nodes. This greatly accelerates the optimization process. To keep the diversity of the population while guiding the selection process towards a good spread of solutions on the Pareto curve, we use similar crowding heuristics to previous work [Deb et al. 2002]. We use crossover to combine partial solutions from high-fitness variants, along with mutation to avoid local optima. In particular, two rendering settings swap their parameter values to generate two offspring. Newly generated variants are considered and compared together with all preferred variants using our partial order. Newly preferred variants are selected to form the incoming population for the next iteration. The result of this process is a Pareto frontier defined for each camera.

5.2 Comparing Pareto Frontiers

The next step is to compare the Pareto frontiers of adjacent cameras sharing the same view direction, and evaluate the numerical difference between them, to decide whether the node should be subdivided. Our observation is that these adjacent cameras will cover a similar portion of the scene and produce a similar image, thus having similar Pareto-optimal rendering settings.

Suppose we already have two Pareto frontiers P_{c_0} and P_{c_1} taken from two different cameras c_0 and c_1. We separately measure their difference under both the error and power metrics:

$$D_e(P_{c_0}, P_{c_1}) = d_e(P_{c_0}, P_{c_1}) + d_e(P_{c_1}, P_{c_0}) \quad (4)$$
$$D_p(P_{c_0}, P_{c_1}) = d_p(P_{c_0}, P_{c_1}) + d_p(P_{c_1}, P_{c_0}) \quad (5)$$

where $d_e(P_{c_0}, P_{c_1})$ and $d_p(P_{c_0}, P_{c_1})$ are half-distance functions computing error and power differences, respectively, and $D_e(P_{c_0}, P_{c_1})$ and $D_p(P_{c_0}, P_{c_1})$ are the two full distance functions of error and power.

Figure 5 shows the process of comparing Pareto curves: To compute the half distance from P_{c_0} to P_{c_1}, we project P_{c_0} to the two-dimensional cost space of P_{c_1}. This can be done by using rendering settings of P_{c_0} to render scenes with camera c_1. Note that both error and power change in the projected curve $P^p_{c_0}$, since it is now related to a different view. Then we compute the distance between $P^p_{c_0}$ and P_{c_1}; for efficiency, we compute the point-wise distances and average them to obtain the total distance. Specifically, given a projected rendering setting s^p_{0j} defining a point $(p_{s^p_{0j}}, e_{s^p_{0j}})$, we find the nearest 2D point on P_{c_1}, defined as $n(c_1, s^p_{0j}) = (p'_{s^p_{0j}}, e'_{s^p_{0j}})$. The error and power distances between these two points are $d_e(s^p_{0j}, n(c_1, s^p_{0j})) = |e_{s^p_{0j}} - e'_{s^p_{0j}}|$, and $d_p(s^p_{0j}, n(c_1, s^p_{0j})) = |p_{s^p_{0j}} - p'_{s^p_{0j}}|$, respectively. The total error and power distance function from the projected Pareto frontier $P^p_{c_0}$ to P_{c_1} is given by:

$$d_e(P_{c_0}, P_{c_1}) = \frac{1}{N}\sum_j^N d_e(s_{0j}, P_{c_1}) \approx \frac{1}{N}\sum_j^N |e_{s_{0j}} - e'_{s_{0j}}| \quad (6)$$

$$d_p(P_{c_0}, P_{c_1}) = \frac{1}{M}\sum_j^M d_p(s_{0j}, P_{c_1}) \approx \frac{1}{M}\sum_j^M |p_{s_{0j}} - p'_{s_{0j}}| \quad (7)$$

where N and M are numbers of rendering settings on P_{c_0} and P_{c_1} respectively (we have removed the super-index p in s_{0j} to simplify notation). If the distances of either the error or power between two Pareto frontiers are larger than a given threshold, we adaptively subdivide our space, as explained in the following subsection.

5.3 Adaptive Space Subdivision

Since computing and comparing all the Pareto curves in the entire camera view-space is intractable, we adaptively partition this space into an octree, storing a set of discrete position-view pairs, according to the distance between Pareto frontiers. Let us consider the position-view pairs, i.e. two cameras (o_1, v_2) and (o_2, v_2) illustrated (as a 2D quadtree) in Figure 3.a, right. If either the error or the power difference between their Pareto frontiers is larger than a given threshold, this indicates that the current sampling of v_2 for adjacent camera positions o_1 and o_2 is insufficient to obtain all power-optimal settings in the space in-between, and the node needs to be subdivided. At the newly generated corner point (o_7 in Figure 3.a, right), we take (o_7, v_2) (blue) as a new camera and compute a Pareto frontier on it, iteratively repeating the comparison-subdivision steps until no more subdivision is required (adjacent Pareto curves are similar). Note that for other views at the corner point o_7, new optimization are only required on the views whose parent views differ above the given threshold; the rest of the views simply inherit one of the Pareto curves of their parent nodes. For views of corners at the center of node faces or at the node center that have more than two adjacent corners, e.g. o_5 and o_{10} in Figure 3.a, we pair their adjacent corners along axes and calculate the corresponding error or power difference. If the difference is larger than the threshold, we then perform optimization on it to compute new Pareto frontier.

6 Runtime Rendering

At run-time, we leverage our adaptively partitioned camera view-space with the corresponding optimal rendering settings to ensure rendering power-optimal images. Observe a 2D quadtree example in Figure 3b. First, given a new camera position and user view, we traverse the octree to obtain the leaf node where it is located. We project the user's frustum onto the cubemap formed by the sides of the leaf, and select the side with the largest projected portion of the view frustum v_2 (see Figure 3b left, green). The corner closest to the camera position o_7, and the selected view v_2 determine a position-view camera sample (o_7, v_2), from which we fetch the precomputed Pareto frontier (see Figure 3b, right). Finally, given a power or error budget, and the selected Pareto frontier, we perform binary search along the frontier and obtain the optimal settings that are used to render the image.

Temporal Filtering of Rendering Settings To avoid visible sudden changes in quality when choosing different cameras during real-time navigation of the scene, we apply a smoothing strategy based on temporal filtering. The filtered rendering settings are computed as

$$s_{\text{optimal}} = [(1 - \frac{t}{T})s_{\text{old}} + \frac{t}{T}s_{\text{new}}] \quad (8)$$

where the brackets denote the closest integer, s_{old} and s_{new} are the previous and current optimal rendering settings, respectively, t is time after applying a new rendering setting, and T is the time used for interpolation ($T = 2$ seconds as default).

7 Implementation

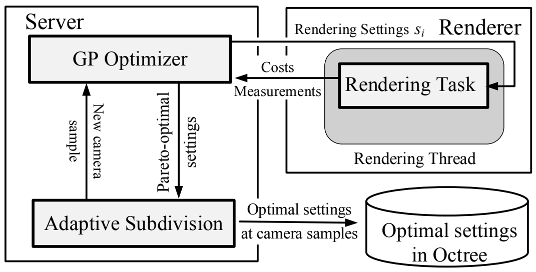
We have implemented an OpenGL-based rendering system, and tested it on two different platforms: One is a desktop PC with Intel Xeon E3-1230 CPU and an NVIDIA Quadro K2200 graphics card, running Microsoft Windows 7. The other is a smartphone with 2.2 GHz 8-core ARM Cortex-A53 CPU and PowerVR G6200 GPU, running Android 5.0.2. Additionally, we also validate our approach on a commercial rendering engine by integrating it into the Unreal Engine [UnrealEngine 2015] on the desktop PC. Please refer to the supplementary material for more details not covered in this section. Our code will be made available through our website.

7.1 Power Measurement

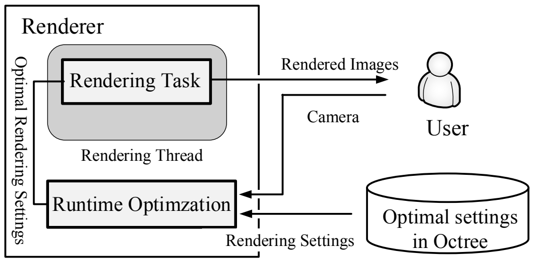
We first set our rendering system to a fixed frames-per-second rate, to guarantee comparable measurements where the only variable are the rendering settings. Then, we combine rendering settings from different cameras, following two different strategies according to the given platform.

Desktop PC To measure the power usage of the Quadro graphics card, we use the C-based API, NVIDIA Management Library (NVML) [NVML 2015], to directly access the power usage of the GPU and its associated circuitry. According to the documentation, measurements are accurate to within $\pm 5\%$ of the current power draw. We average power measurements over a given time period to reduce variance: we generally take 10 seconds to measure the power and read back 10 times per second. Between two different rendering settings, we wait for 3 seconds without measurements to avoid any residual influence of the previous setting.

Mobile Device For the smartphone, we use an external source meter to directly supply the power of the device. The source meter



(a) *Adaptive subdivision*



(b) *Runtime Rendering*

Figure 6: *Preprocess and runtime workflow of our system. Please refer to the text in Section 7 for details.*

that we are using is a Keithley A2230-30-1, which allows direct cable connection and provides APIs to access the instantaneous voltage and current consumption. In practice, we set a constant voltage and read back the current, from which we obtain power. Note that in this case, both the CPU and GPU power usages are measured. Before the measurement, we close all unnecessary applications and services to reduce the unpredictable power consumptions of CPU. Since the power measurement of the mobile device has bigger variance than the desktop PC, we average over 25 seconds, and read back 10 times per second. The interval between the measurements of two rendering settings is 5 seconds.

7.2 Rendering Systems

7.2.1 OpenGL-based Rendering System

Figure 6 shows the architecture and the workflow of our rendering system. It consists of a renderer and a server, connected through sockets. The renderer is developed in C++ and OpengGL ES, to be easily used on different platforms. The server is implemented in C++ and only executed on the desktop PC. Our system has two rendering modes: In the *subdivision* mode, the renderer receives information from the server about the camera position and view to render scenes, to perform the adaptive partition of the view space. After the preprocess, all measured and sampled data are then transferred from the server to the renderer. The *rendering* mode is active during free navigation of the scene. The renderer automatically searches in the stored hierarchy to find the power-optimal rendering settings at run-time.

Rendering settings Our OpenGL rendering framework supports GPU-based *importance sampling* [Colbert and Křivánek 2007], *shadow mapping*, *screen-space ambient occlusion* (SSAO) [Kajalin 2009], and *morphological antialiasing* (MLAA) [Jimenez et al. 2011]. For each, we can choose between different parameters and values to adjust the rendering quality, resulting in a varying power consumption. The combination of all these makes up the space of all rendering settings. For GPU-based importance sampling, the parameter we use is the number of samples generated at run-time; for shadow mapping, we choose the shadow map resolution as parameter; for SSAO, the parameter is the number of sample rays to compute the visibility; last, for MLAA, we vary the steps of the search to find edges in the pixel shader. The complete set of values is given in Table 1. To store these settings we use a 32-bit integer, where the index value of each effect takes up eight bits.

Parameters	Values
In-house renderer	
Sample number in GPU sampling	8, 16, 32
Shadow map resolution	256, 512, 1024, 2048
Sample number in SSAO	4, 8, 16, 32
Search steps in MLAA	4, 8, 16, 32, 64
The renderer in Unreal Engine	
Anti aliasing	0, 2, 4, 6
Post processing quality	0, 1, 2, 3
Shadows quality	0, 1, 2, 3
Textures quality	0, 1, 2, 3
Effects quality	0, 1, 2, 3
Resolution scale	70%, 80%, 90%, 100%
View distance	0.1, 0.4, 0.7, 1.0

Table 1: *List of parameters and values forming the space of rendering settings for our two renderers. For details in the Unreal Engine parameters, refer to the supplementary material.*

Adaptive Subdivision In subdivision mode, the server sends camera information and rendering settings to the renderer, to sample the camera view-space. For each sample, the server first requests the renderer to render the scene with maximum quality and store the image as a reference, to be used to compute quality errors. Then, the server runs the Genetic Programming algorithm to optimize the Pareto frontier. The server sends each of the candidate rendering settings to the renderer, and let the renderer use it to render the scene. The power usage is measured by the server, and the image error is measured by the renderer and sent back to the server. In the next step, the server compares the Pareto frontiers and selects the next camera sample. The process is repeated iteratively until no more subdivisions are needed. Last, the server stores all the Pareto frontiers at the views of corners of the final octree.

Runtime Rendering In the real-time rendering mode, the position and view of the camera are used to guide the search in the octree. Then, the power-optimal rendering setting is retrieved and used to render the scene.

7.2.2 Rendering System Using the Unreal Engine

To test how well our framework generalizes to other rendering platforms, we implement it on the Unreal Engine. This framework also consists of two sub-systems, the renderer and the server, with similar roles as before. To integrate our rendering in the Unreal Engine, instead of defining two modes of operation, we develop two plugins for the subdivision and rendering tasks, respectively, adapting our in-house code to the Unreal architecture.

Rendering settings The Unreal Engine provides a set of predefined settings to allow users to adjust the quality of several features. These can be tweaked at run-time, thus they fit well in our system. We select seven features (Table 1): *resolution scale*, *view distance*, *anti-aliasing*, *post-processing* quality, *shadows* quality, *textures* quality, and the *effects*. The complete set of values is given in Table 1, defining the space of all rendering settings. To store these settings we also use a 32-bit integer, where the index value of each effect takes up four bits.

Demos	Renderer	Platform	Scene		Subdivision						Rendering
			Tris. #	Scene Size	Thld. (power, error)	Time (hrs)	Levels	Corners	Views	Data Size	Duration
Valley	in-house	PC	55.9 k	32.5 MB	(0.5W, 0.002)	29.9	4	279	759	18.1 KB	50 s
Hall	in-house	Mobile	31.7 k	25.8 MB	(0.15W, 0.002)	35.3	4	108	316	9.3 KB	50 s
Elven Ruins	UE4	PC	191.2 k	1.03 GB	(0.5W, 0.005)	67.1	5	393	956	36.6 KB	60 s
Sun Temple	UE4	PC	667.9 k	512 MB	(0.5W, 0.005)	35.4	3	125	511	18 KB	60 s

Table 2: *Statistics for the tested demos. Scene statistics include number of triangles, and size on disk. Subdivision statistics include number of levels and corners on the octree, number of computed views, size on disk, and preprocess time. Thresholds indicate power and error limits for subdivision of the octree (see Section 5 for details).*

8 Results

We performed a series of experiments in order to demonstrate the effectiveness of our rendering framework on four different scenes. Our in-house renderer runs on a desktop PC and a smartphone, and the renderer integrated in the Unreal Engine runs on the desktop PC. In Table 2, we summarize the statistics of the demo scenes. The FPS is set to 30 in all our experiments on desktop PC, and 10 on the smartphone due to the limited computational power.

Valley We use our in-house renderer to render the scene with four different effects on the PC. We set an environment light and a directional light, with GPU-based importance sampling. The directional light casts shadows, with the shadow map resolution as one of the parameters in our optimization. The screen-space ambient occlusion (SSAO) and morphological anti-aliasing (MLAA) are computed as post-processes.

Hall Each polygon has a diffuse map and a specular map. We use our in-house renderer to render the scene on the smartphone. Since the GPU-based importance sampling effect requires environment lighting, we initially render the scene onto a cubemap centered in the hall, and use it as the environment map. A directional light is set to illuminate the scene through the door. As in the previous scene, SSAO and MLAA are all computed in screen space as post-processes.

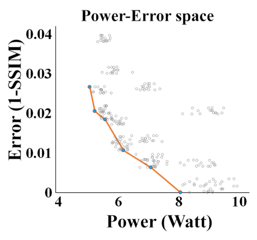
Elven Ruins This demo is modified from an example scene shipped with the Unreal Engine. We use the plug-in that we developed in the Unreal Engine to render the scene.

Sun Temple This demo is another example scene shipped with the Unreal Engine. We also use our Unreal Engine plug-in to render it.

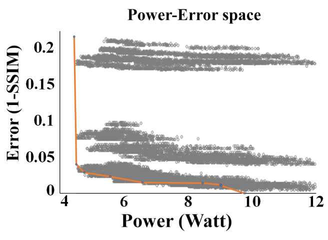
8.1 Adaptive Subdivision and Pareto Frontier

Power and error thresholds used to trigger subdivision of the octree are shown in Table 2. Note that since the parameter space is different for our in-house renderer and Unreal Engine, while power consumption also varies on each platform, we set different initial parameters for each platform-renderer pair. For the Genetic Programming (GP) algorithm, we set the maximum iterations to 25 in our in-house renderer. In Unreal Engine, we increase the maximum iterations to 40 due to the higher complexity of the parameter space. As Table 2 shows, the extra memory overhead is negligible in both cases, in the order of a few KB.

Figure 7 shows two example plots of our entire power-error cost space for one view in the *Valley* and *Elven Ruins* scenes. The Pareto frontiers optimized by our GP algorithm are shown in orange. The combinations of all different rendering settings are shown in dark grey.

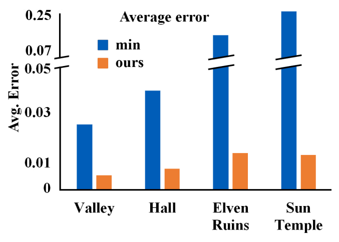


(a) *One view in* Valley *demo*

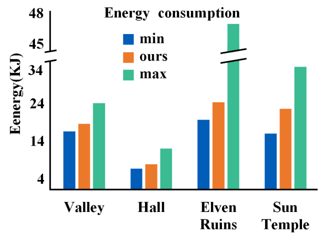


(b) *One view in* Elven Ruins *demo*

Figure 7: *The entire power-error cost spaces and Pareto frontiers optimized by our GP algorithm at two example views. Grey dots are rendering settings and the orange line is the Pareto frontier. (a) One view in the* Valley *demo with 300 rendering settings. (b) One view in the* Elven Ruins *demo with 16384 rendering settings.*



(a) *Average visual error*



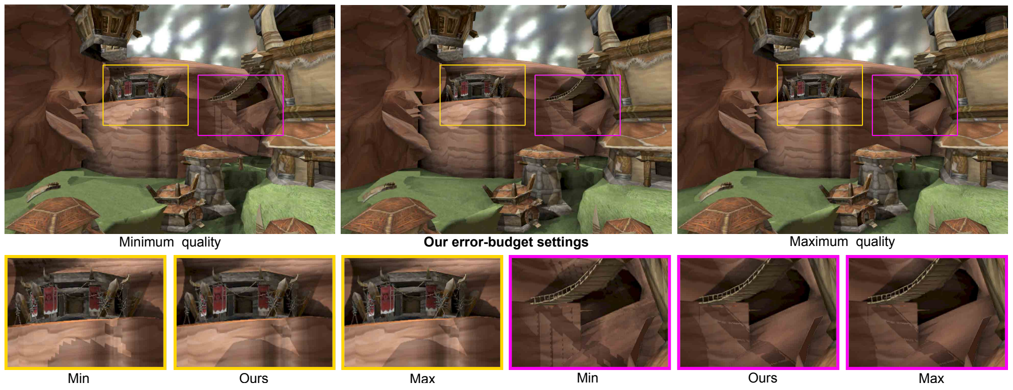
(b) *Energy consumption*

Figure 8: *Average visual error and total energy consumption under different rendering settings in our four demos.*

8.2 Runtime Power-Optimal Rendering

Although our approach supports run-time free exploration, for comparison purposes we record a camera path and repeat the motion while testing different power or error budgets. To obtain stable reliable measurements, all paths last between 50 and 60 seconds. The maximum quality and the minimum rendering settings are regarded as the baselines. Then, for the different demos, we use different power or error budgets to guide our run-time power-optimal rendering. Figure 8 shows the average visual error and total energy consumption we measured. It can be seen how our framework drastically minimizes visual error, while keeping power consumption very close to the minimum-quality settings. We describe our demos in this section, and refer the reader to the supplemental video for the animations and the details of all the rendering settings.

Figure 1 shows the *Sun Temple* scene running on the Unreal Engine, under a power budget $p = 7W$. From the zoomed-in insets, it can be clearly seen how the quality produced with our framework is very close to the maximum quality, while the minimum rendering settings introduce visible artifacts such as wrong shadows, over-blurred areas, or missing reflections. The plots on the right show a lower power usage than maximum quality, with negligible error.



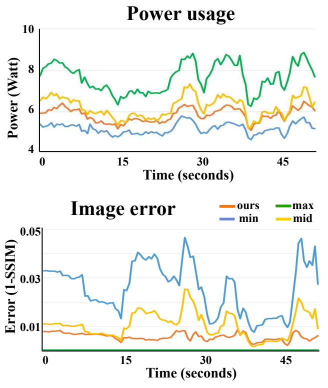
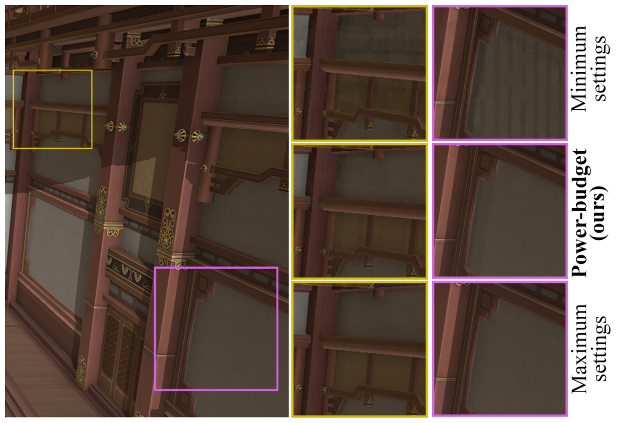


Figure 9: *Real-time demo for the* Valley *scene, using our in-house render engine on a desktop PC. We compare minimum, middle and maximum quality settings against our power-optimal rendering framework, selecting an error threshold of 0.01 (renderings with middle settings not shown in the figure). Our method outputs images almost identical to those produced with the maximum-quality settings. The plots on the right show power usage and error during 50 seconds of free camera navigation: Our framework keeps power consumption almost as low as the minimum settings (top), while ensuring that errors stay below the given threshold. Please refer to the supplementary video for the full demo.*



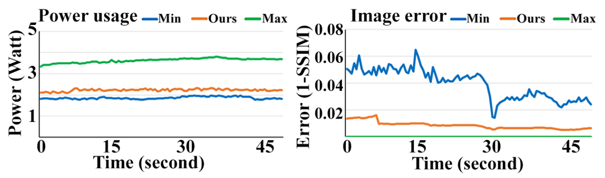


Figure 10: Hall *real-time demo using our in-house render engine on a smartphone. Top-left image shows the maximum settings render. We compare insets (top-right) of minimum and maximum quality settings against our power-optimal rendering framework, selecting a power threshold of 2.2W, close to consumption at the minimum quality settings. Plots on the bottom show power usage and error during a 50-second camera path. Our optimized settings maintain a power usage below the given budget, while providing a quality close to the maximum settings. Please refer to the supplementary video for the full demo.*

Figure 9 shows the *Valley* scene rendered on a desktop PC. During navigation under a visual error budget $e_{\text{bgt}} = 0.01$, our system automatically retrieves the optimal rendering settings to produce the image in real-time. We measure and compare, for the maximum and minimum rendering settings, the power usage and the error curves, plotted in Figure 9, right. Since this scene is relatively simple, the error between the maximum and minimum rendering settings is not extremely large. But even in this case, our method is able to find the optimal tradeoffs that keep the error within budget, while significantly reducing power usage.

Elven Ruins					
	Ours	Max	Max-Mid	Mid-Min	Min
Avg. Power (W)	7.87	15.87	11.80	8.88	6.11
Avg. Error (e)	0.018	0	0.013	0.046	0.151

Sun Temple					
	Ours	Max	Max-Mid	Mid-Min	Min
Avg. Power	7.09	10.73	8.51	6.00	4.90
Avg. Error	0.0145	0	0.0158	0.068	0.236

Table 3: *Average power consumption and error for the* Elven Ruins *and* Sun Temple *demos, using different rendering settings. Our power optimal framework (Ours), achieves the best tradeoff, producing images almost identical to the maximum quality settings while reducing power between 30% and 50% approximately.*

Figure 10 shows a detail of the *Hall* demo running on the smartphone. We set a power budget $p = 2.2W$. Three power usage curves and error curves are plotted in Figure 10, bottom. Our method stays within the power budget, offering a good tradeoff between power and error.

Figure 11 shows the *Elven Ruins* demo, running on the Unreal Engine. Here we set two different budgets, a power budget $p = 10W$, and a visual error budget $e = 0.02$. Our system takes these two budgets into account during navigation, and selects power-optimal rendering settings accordingly. The plots on the right show how, if we use the power budget to guide the rendering, power consumption is more stable than using the error budget. In this case, our framework dynamically finds some rendering settings that dramatically reduce power consumption, bringing it close to the minimum consumption (at 25-32s and 48-60s), while maintaining a very low quality error. This is because our system automatically identifies which rendering parameters have a larger impact on quality for the current view, while still maintaining low power consumption.

8.3 Analysis of Different Settings

A key advantage of our optimization framework is its flexibility, being agnostic to the particular choice of parameters and settings. This is a key feature, since we have not found a predictable correlation between the values for the different parameters and their effect on power saving and error. This impact is instead highly dependent on the particularities of the scene and view being rendered. For instance, Shadow Quality will only have a measurable effect when shadows are clearly visible in the frame (see for instance Figure 9).

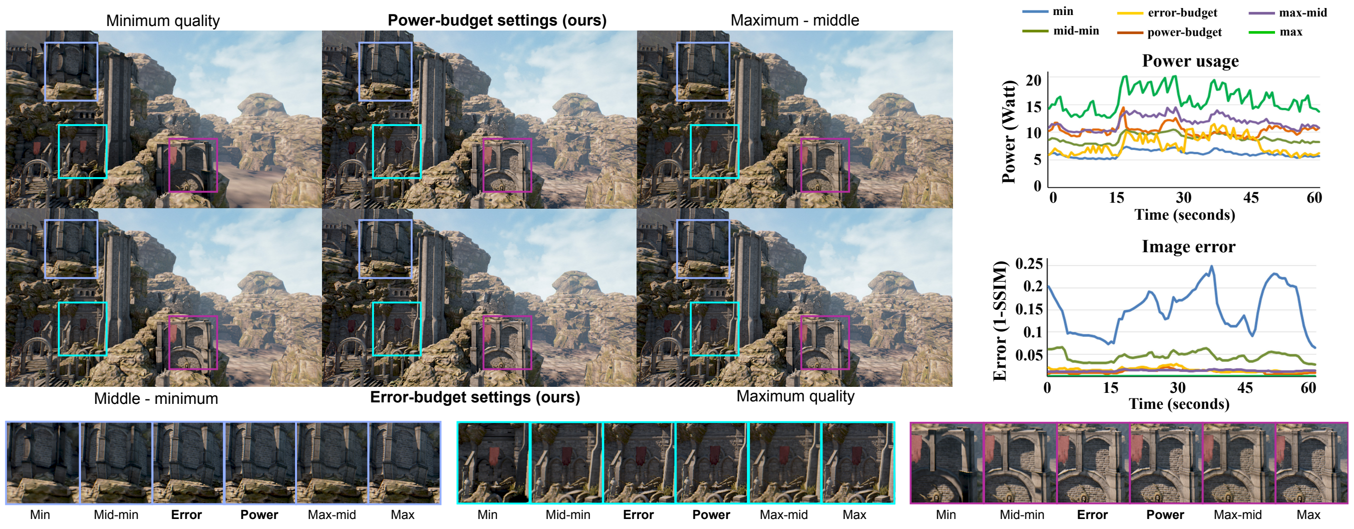


Figure 11: Elven Ruins *real-time demo using the Unreal Engine on a desktop PC. We compare minimum, middle-minimum, maximum-middle and maximum quality settings against two modes of our power-optimal rendering framework: Selecting a power budget of 10W, and an error budget of 0.02. Plots on the right show power usage and error during a 60-second camera path. Our framework provides power-error optimized settings under the two different budget modes, while guaranteeing in both cases a visual quality very similar to the maximum settings, and power usage close to the minimum settings. Moreover, our optimized settings outperform manual settings. Please refer to the supplementary video for the full demo.*

The only exceptions to this for our parameter space are Resolution Scale, which has a direct correlation with power consumption, and Texture Quality, which in our tests seemed to impact image quality the most. However, even these two parameters have a very different influence on power and error depending on the rendered view, as Figure 12 shows. Given the entire camera view-space of a scene, it would obviously be impractical to manually preset all optimal rendering settings. Our framework allows us to automatically select optimal power and error settings at runtime, without human intervention of prior knowledge about the scene.

We have also conducted a comparison with manually set tradeoffs between power and quality. In the Unreal Engine, some settings can be manually tweaked, allowing users to adjust the quality of various features. We set settings for four quality levels: maximum (all values set to maximum), maximum-middle, middle-minimum, and minimum (all values set to minimum). For this test we use the *Elven Ruins* and the *Sun Temple* scenes. Figure 11 compares one view under different settings and the corresponding plots for power and error in the *Elven Ruins* scene. The statistics of average power usage and errors are shown in Table 3. As can been seen, our method provides an excellent balance between visual error and power consumption: In the *Elven Ruins* demo, our method only consumes 7.87 W, which represents a saving of 50.4% of power compared to the maximum setting, and the visual quality is an order of magnitude better than the minimum setting. Similar conclusions can be inferred for the *Sun Temple* demo, with about 30% less power consumed. These results clearly demonstrate that our power-optimal framework is capable of automatically balancing optimal power consumption and quality, which would be very challenging to achieve by manually adjusting settings. Moreover, our framework provides *dynamic* optimal settings, while manually-set parameters in Unreal remain fixed throughout the demo.

8.4 Temporal Filtering

As described in Section 6, to reduce sudden changes in quality due to the runtime optimization of rendering settings, we apply a temporal filtering strategy. Despite this filtering being a discrete interpolation, our simple smoothing strategy improves the rendering quality in many cases. Figure 13 shows an example of 200 frames, including a runtime change of parameters with and without temporal filtering. We use the parameters before this change to render 200 frames, and regard them as reference to compute visual error. As shown in the plot, our temporal filtering provides a smoother transition, gradually modulating the visual error after a parameter change at frame #35, successfully reducing visible popping artifacts (refer also to the video in the supplemental). The zoomed insets of frame #34 and #35 clearly demonstrate better consistence when applying the temporal filtering.

9 Discussion and Future Work

In some cases, the power or error curves may deviate slightly from the given budgets. This is due to the following reasons: First, at each view, the Pareto-optimal settings are discretely distributed in the power-error space. Therefore, the optimal setting computed under a budget may not exactly match the budget. Second, during the adaptive subdivision, the camera view-space is partitioned by thresholds until a fixed number of octree levels is reached. Therefore, in some local regions, using the optimal rendering setting at the closest sample camera may induce a slight deviation. In any case, as shown in Figure 11, the error and power curves remain very stable.

Since we focused on optimizing GPU consumption, we explicitly measured GPU power on desktop PC, and minimized CPU impact on mobile devices by deactivating as many external CPU sources as possible. While some rendering aspects may influence CPU power usage, in practice we found this variation negligible for the parameters we used. Nevertheless, it remains an interesting topic of future work to analyze the influence of a wider set of parameters on CPU power usage.

Our framework is not free of limitations and potential avenues of future work. First, it does not take into account dynamic changes in geometry or lighting. However, predicting the full space of all possible situations that may arise when dynamic changes area

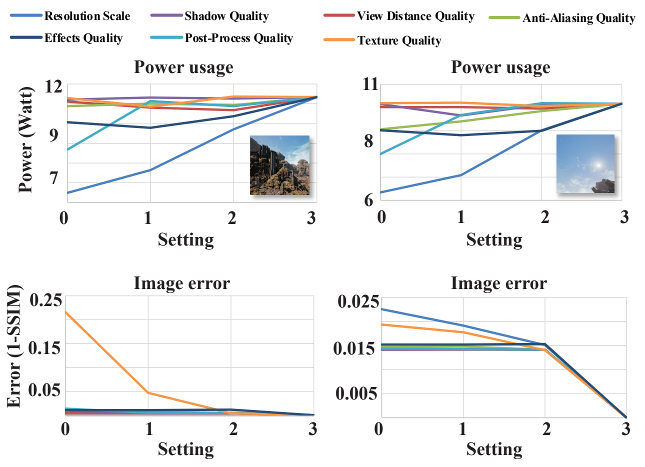


Figure 12: *The influence of the different parameters is highly dependent on the particularities of the scene and view being rendered. Shown are two views of the* Elven Ruins *demo. For each view, we first set all parameters to produce the maximum rendering quality, and use it as the base setting. Then, we individually change only one parameter, from minimum to maximum value (shown as 0..3 in the figure), while keeping the others at maximum level. From the power consumption plots, it can be seen that in this case Resolution, Effects and Post-Process are the most dominant. However, the error is inversely correlated with Texture quality for the first view, while Resolution has an insignificant impact.*

allowed is obviously intractable. One possibility to incorporate such changes in our optimization would be to precompute some of them, for instance the same view under different illuminations, and smoothly interpolate between settings at runtime. Our framework allows for such extensions of the power-error cost space, at the cost of longer preprocessing times. Nevertheless, this would only need to be done once; at run-time, the system would still be able to optimize in real-time, given our strategy of reducing the search for optimal settings to a one-dimensional Pareto curve.

Second, the capability to explore the full space of all possible combinations of rendering settings is limited by our GP optimization. Different strategies may yield slightly different Pareto frontiers, although we do not expect the final results to vary much in terms of power consumption or visual quality during navigation. Similarly, we have set our thresholds for the adaptive subdivision heuristically: although they provide a good balance between complexity of the subdivision and performance, we did not thoroughly explore the possibilities of other subdivision thresholds or schemes.

While the results in this paper are strictly valid for the specific hardware configuration we used, our proposed framework is equally applicable to any other configurations. Moreover, we believe that the resulting optimization for a particular hardware setup will allow for a certain degree of transferability across similar configurations, by abstracting some of the dependencies. Finally, although the required precomputation time is not significant for large-scale productions, it would be interesting to find novel ways to reduce it, for instance by learning relationships among scene properties, rendering parameters and power usage, or acquiring higher-level knowledge about parameters.

To summarize, we hope that our power-saving rendering framework inspires future work in this direction. Our current implementation satisfies four key ideal characteristics: it produces optimal results between energy consumption and quality; it allows the user to fix

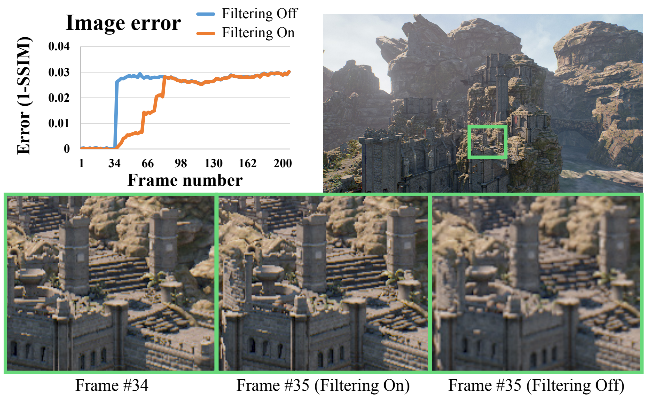


Figure 13: Elven Ruins *scene with and without our temporal filtering, for a parameter change at frame #34. To illustrate the benefits, we propagate the settings at frame #34 to 200 frames, and compute the error of each frame with respect to the filtered (on) and non-filtered (off) versions. Notice how temporal filtering improves consistency, avoiding visible popping artifacts (sudden jump in the orange curve) by providing a smoother transition between settings. This is also shown in the insets comparing frames #34 and #35.*

either a power or a quality target; it is real-time; and it generalizes across different platforms. We have shown results on four different scenes, running on two different platforms, including a commercial one. Additionally, we have validated that our framework outperforms manually-set parameters, available in the Unreal Engine environment.

Acknowledgements

We would like to thank all reviewers for their thoughtful comments. We also thank Jiali Pan, Yunjing Zhang and Xianlong Ge for preparing demos and video, Carlos Aliaga and Elena Garces for helping with figures, and Adrian Jarabo and Adolfo Muñoz for proofreading the paper. This work was partially supported by 973 Program of China (No. 2015CB352503), NSFC (No. 61472350), the Fundamental Research Funds for the Central Universities (No. 2016FZA5013), the European Social Fund, the Gobierno de Aragón, and the Ministerio de Economía y Competitividad (project LIGHTSLICE).

References

AKENINE-MÖLLER, T., AND STROM, J. 2008. Graphics processing units for handhelds. *Proceedings of the IEEE 96*, 5 (May), 779–789.

ARNAU, J.-M., PARCERISA, J.-M., AND XEKALAKIS, P. 2014. Eliminating redundant fragment shader executions on a mobile GPU via hardware memoization. *SIGARCH Comput. Archit. News 42*, 3 (June), 529–540.

BELOGLAZOV, A., ABAWAJY, J., AND BUYYA, R. 2012. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Gener. Comput. Syst. 28*, 5 (May), 755–768.

CHEN, H., WANG, J., CHEN, W., QU, H., AND CHEN, W. 2014. An image-space energy-saving visualization scheme for OLED displays. *Computers & Graphics 38*, 61 – 68.

CHEN, W., CHEN, W., CHEN, H., ZHANG, Z., AND QU, H. 2016. An energy-saving color scheme for direct volume rendering. *Computers & Graphics 54*, 57 – 64.

CHENG, W.-C., AND PEDRAM, M. 2004. Power minimization in a backlit TFT-LCD display by concurrent brightness and contrast scaling. *IEEE Transactions on Consumer Electronics 50*, 1, 25–32.

CHUANG, J., WEISKOPF, D., AND MLLER, T. 2009. Energy aware color sets. *Computer Graphics Forum 28*, 2, 203–211.

COHADE, A., AND DE LOS SANTOS, S. 2015. Power efficient programming: How funcom increased play time in lego minifigures. In *Game Developer's Conference*.

COLBERT, M., AND KRIVÁNEK, J. 2007. GPU-based importance sampling. *GPU Gems 3*, 459–476.

DEB, K., PRATAP, A., AGARWAL, S., AND MEYARIVAN, T. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *Trans. Evol. Comp 6*, 2, 182–197.

DONG, M., AND ZHONG, L. 2012. Power modeling and optimization for oled displays. *IEEE Transactions on Mobile Computing 11*, 9, 1587–1599.

DONG, M., CHOI, Y.-S. K., AND ZHONG, L. 2009. Power-saving color transformation of mobile graphical user interfaces on oled-based displays. In *Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '09, 339–342.

FORREST, S. R. 2003. The road to high efficiency organic light emitting devices. *Organic Electronics 4*, 2 - 3, 45 – 48.

GHARBI, M., SHIH, Y., CHAURASIA, G., RAGAN-KELLEY, J., PARIS, S., AND DURAND, F. 2015. Transform recipes for efficient cloud photo enhancement. *ACM Trans. Graph. 34*, 6 (Oct.), 228:1–228:12.

HE, Y., FOLEY, T., TATARCHUK, N., AND FATAHALIAN, K. 2015. A system for rapid, automatic shader level-of-detail. *ACM Trans. Graph. 34*, 6 (Oct.), 187:1–187:12.

IYER, S., LUO, L., MAYO, R., AND RANGANATHAN, P. 2003. Energy-adaptive display system designs for future mobile environments. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, 245–258.

JIMENEZ, J., MASIA, B., ECHEVARRIA, J. I., NAVARRO, F., AND GUTIERREZ, D. 2011. GPU Pro 2.

JOHNSSON, B., GANESTAM, P., DOGGETT, M., AND AKENINE-MÖLLER, T. 2012. Power efficiency for software algorithms running on graphics processors. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, EGGH-HPG'12, 67–75.

JOHNSSON, B. M. 2014. *Energy Analysis for Graphics Processors using Novel Methods & Efficient Multi-View Rendering*. PhD thesis, Lund University.

KAJALIN, V. 2009. Screen space ambient occlusion. *Shader X 7*, 413, 24.

KYUNG, C.-M., AND YOO, S. 2014. *Energy-Aware System Design: Algorithms and Architectures*. Springer Publishing Company, Incorporated.

MASIA, B., WETZSTEIN, G., DIDYK, P., AND GUTIERREZ, D. 2013. A survey on computational displays: Pushing the boundaries of optics, computation, and perception. *Computers & Graphics 37*, 8, 1012–1038.

MAVRIDIS, P., AND PAPAIOANNOU, G. 2015. MSAA-based coarse shading for power-efficient rendering on high pixel-density displays. In *High Performance Graphics*.

MOSHNYAGA, T., AND MORIKAWA, E. 2005. LCD display energy reduction by user monitoring. In *IEEE international conference on computer design*.

NARRA, P., AND ZINGER, D. 2004. An effective LED dimming approach. In *IEEE industry applications conference*.

NVML, 2015. NVIDIA Management Library. https://developer.nvidia.com/nvidia-management-library-nvml.

PEDDIE, J. 2013. Trends and forecasts in computer graphics – power-efficient rendering. In *Jon Peddie Research*.

PELLACINI, F. 2005. User-configurable automatic shader simplification. *ACM Trans. Graph. 24*, 3 (July), 445–452.

POOL, J. 2012. *Energy-precision tradeoffs in the graphics pipeline*. PhD thesis, University of North Carolina at Chapel Hill.

POWERVR. 2012. PowerVR: A master class in graphics technology and optimization. In *Imagination Technologies*.

RANGANATHAN, P., GEELHOED, E., MANAHAN, M., AND NICHOLAS, K. 2006. Energy-aware user interfaces and energy-adaptive displays. *Computer 39*, 3, 31–38.

SHEARER, F. 2007. *Power Management in Mobile Devices*. Elsevier Inc.

SITTHI-AMORN, P., MODLY, N., WEIMER, W., AND LAWRENCE, J. 2011. Genetic programming for shader simplification. *ACM Trans. Graph. 30*, 6, 152.

STAVRAKIS, E., POLYCHRONIS, M., PELEKANOS, N., ARTUSI, A., HADJICHRISTODOULOU, P., AND CHRYSANTHOU, Y. 2015. Toward energy-aware balancing of mobile graphics. In *IS&T/SPIE Electronic Imaging*, International Society for Optics and Photonics, 94110D–94110D.

UNREALENGINE, 2015. Unreal Engine. https://www.unrealengine.com/.

VAIDYANATHAN, K., SALVI, M., TOTH, R., FOLEY, T., AKENINE-MÖLLER, T., NILSSON, J., MUNKBERG, J., HASSELGREN, J., SUGIHARA, M., CLARBERG, P., ET AL. 2014. Coarse pixel shading. In *High Performance Graphics*.

VALLERIO, K. S., ZHONG, L., AND JHA, N. K. 2006. Energy-efficient graphical user interface design. *IEEE Transactions on Mobile Computing 5*, 7 (July), 846–859.

WANG, Z., BOVIK, A. C., SHEIKH, H. R., AND SIMONCELLI, E. P. 2004. Image quality assessment: from error visibility to structural similarity. *Image Processing, IEEE Transactions on 13*, 4, 600–612.

WANG, R., YANG, X., YUAN, Y., CHEN, W., BALA, K., AND BAO, H. 2015. Automatic shader simplification using surface signal approximation. *ACM Trans. Graph. 33*, 6 (Nov.), 226:1–226:11.

WOO, R., YOON, C.-W., KOOK, J., LEE, S.-J., AND YOO, H.-J. 2002. A 120-mW 3-D rendering engine with 6-Mb embedded DRAM and 3.2-GB/s runtime reconfigurable bus for PDA chip. *Solid-State Circuits, IEEE Journal of 37*, 10 (Oct), 1352–1355.