

Adaptive undersampling for efficient mobile ray tracing

Youngwook Kim¹ · Woong Seo¹ · Yongho Kim² · Yeongkyu Lim³ · Jae-Ho Nah³ ·
Insung Ihm¹ 

Published online: 13 May 2016
© Springer-Verlag Berlin Heidelberg 2016

Abstract Aiming to develop an efficient ray tracer for a mobile platform, we present an adaptive undersampling method that enhances the rendering speed by effectively replacing expensive ray-tracing operations with cheap interpolation whenever possible. Our method explores both object- and image-space information gathered during ray tracing to detect possibly problematic pixels. Rays are fired only for these pixels. We also present a postcorrection algorithm that minimizes annoying artifacts inevitably caused by undersampling. Our implementation on a mobile GPU demonstrates that this method can speed up the rendering computation significantly, while retaining almost the same visual quality of the rendering.

Keywords Ray tracing · Mobile platform · Adaptive undersampling · Postcorrection · GPU algorithm

1 Introduction

Despite recent successes in building efficient ray tracers, optimizing the rendering computation remains desirable and even essential when the computing load for a required rendering task is beyond the processing power of available processors. For instance, the QHD resolution (2560×1440) has nowadays become common for mobile phones whose processors are often not powerful enough for full ray tracing in real

time. An effective way of accelerating the ray-tracing computation is adaptive undersampling, which aims to fire fewer than one ray per pixel, thereby minimizing the total number of ray shootings that incur costly ray–object intersections, while introducing only a small reduction in ray-tracing quality. In fact, the idea of adaptive pixel sampling has long been explored in the ray-tracing community, usually in the context of adaptive supersampling, which aims at reducing aliasing artifacts caused by insufficient point sampling. Whether undersampling or supersampling, the major concern is identical in that the goal is to efficiently detect image-space pixels and/or object-space surface regions that may create aliasing, then adaptively dispatching rays only where necessary, and applying cheaper interpolation whenever possible.

In this paper, we present an adaptive undersampling technique that is well suited for effective implementation of a mobile GPU ray tracer. Our method collects various pixel attributes on the fly during rendering, which are then used to decide, through similarity checks, whether the expensive ray-tracing operations may be replaced by much cheaper linear interpolation for computing geometric attributes at the first hit points (see Fig. 1). Compared to previous adaptive sampling techniques that exploit both image- and object-space information [1, 4, 5, 13], our method is more “geometric” in that it also examines the higher-order local geometry of object surfaces, such as convexity. This reduces the likelihood of subtle visual artifacts that are hard to eliminate using previous methods. In addition, we propose a low-cost postcorrection method that effectively reduces the occurrence of aliases such as the “missing objects” caused by incomplete ray sampling in undersampled images.

The proposed method is simple in structure and easily mapped to the mobile GPU architecture, offering an efficient parallel undersampling computation. In particular, while most of the existing adaptive methods recursively sub-

✉ Insung Ihm
ihm@sogang.ac.kr

¹ Department of Computer Science and Engineering, Sogang University, Seoul, Republic of Korea

² NCSOFT, Seongnam, Republic of Korea

³ LG Electronics, Seoul, Republic of Korea

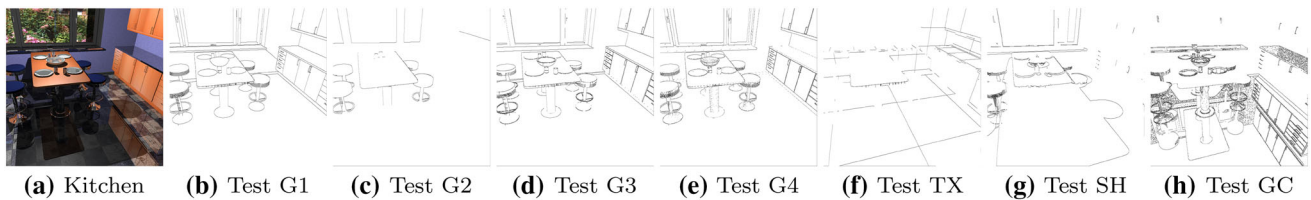


Fig. 1 Problematic adaptive pixels found by our adaptive undersampling method. To render the images (a), our mobile GPU ray tracer performed costly ray-tracing operations only for those problematic pixels that were detected through the seven similarity checks defined in Table 1, as respectively depicted in (b–h). Only 34.5 % of the image

divide pixels for further sampling based on the attributes of four reference corner pixels, our adaptive undersampling algorithm shades a pixel, through ray shooting or interpolation, with reference to only two neighboring pixels. This simple, two-level pixel sampling technique is computationally simpler and requires less memory bandwidth. Therefore, compared with recent adaptive sampling methods such as [8] that are optimized for high performance GPUs, our method will allow more efficient implementation on mobile GPUs, which are more vulnerable to control-path complexity and heavy memory accesses than PC-based GPUs.

2 Previous work on adaptive ray sampling

Adaptive sampling in spatial and temporal spaces has been an important research topic in the ray-tracing community. In his seminal paper, Whitted proposed using hierarchical adaptive supersampling to reduce aliases resulting from the undersampling of high-frequency signals, where pixels were recursively subdivided for further sampling only if colors sampled at their four corners vary significantly [20]. For optimal supersampling in multidimensional space, Lee et al. derived a relationship between the number of ray samples and the quality of the rendering image [10]. Also, for optimal stochastic sampling, Dippé and Wold adaptively determined the sampling rate and filter width based on their error estimates [3]. As a variance reduction technique for solving the rendering equation, Kajiya proposed applying an adaptive hierarchical sampling method so that samples were concentrated in interesting parts of the rendering domain [9].

In his distributed ray-tracing paper, Cook gave an example of using two levels of sampling densities in which a higher-density pattern was applied for troublesome areas [2]. Mitchell also presented a two-level sampling method by subdividing pixels into small squares and finding those that need high-density sampling [12]. Painter and Sloan applied hierarchical adaptive stochastic sampling that worked in a progressive manner [15]. Levoy proposed an adaptive

pixels, including both base and adaptive pixels, were ray traced to create the 1024×1024 image, which were very difficult to distinguish visually from the fully ray-traced image. In general, the ratios of adaptive pixels that fail the respective similarity checks vary in a complicated manner, depending on the scene complexity and rendering parameters

sampling method for volume rendering that also determined the sample rate progressively [11]. Rigau et al. exploited a family of discrimination measures, called the f -divergences, to determine the adaptive sampling rate [16]. Hachisuka et al. proposed a kd -tree-based adaptive refinement and anisotropic integration algorithm for multidimensional sampling in ray tracing [6].

In addition to the color measure, object space information has also been exploited by Thomas et al. [18] and Ohta and Maekawa [14]. Whitted's adaptive sampling scheme [20] was also extended by Genetti et al. so that decisions regarding extra sampling were made based on object-space information obtained during the ray-object intersection computation [5]. Akimoto et al. proposed a four-level undersampling technique, called pixel-selected ray tracing, to speed up the rendering computation [1], in which both image-space and object-space measures were utilized for adaptive ray tracing. Their idea was then extended by Murakami and Hirota [13] and Formella et al. [4]. Jin et al. also presented a selective and adaptive supersampling method, optimized for today's many-core processors [8].

In the context of the rasterization-based rendering pipeline, He et al. [7] and Vaidyanathan et al. [19] independently proposed rendering architectures supporting varying shading rates, where different levels of pixel sampling were adopted to reduce the fragment shading cost. These multi-rate shading methods are similar to ours, in that GPU-oriented, simple structured mechanisms are employed to perform expensive shading operations, ray tracing in our case, only where needed, eventually leading to effective undersampling. However, the rasterization-based approaches are not extendable for developing a GPU ray tracer.

3 Adaptive undersampling algorithm

3.1 Partition of image pixels

Figure 2 shows an example of pixel partitioning, where a set of regularly distributed pixels, marked as B, forms a group of

Fig. 2 Image pixel partitioning through 2×2 base blocks. The pixels marked as B, A1, and A2 represent base pixels, and vertical and horizontal adaptive pixels, respectively

B	A1	B	A1	B
A2	A2	A2	A2	A2
B	A1	B	A1	B
A2	A2	A2	A2	A2
B	A1	B	A1	B

base pixels. The other pixels, called adaptive pixels, are classified as either A1 or A2 depending on whether they are in the same row as the base pixels or not. Previous related methods [1] often traverse pixels in a recursive, multilevel fashion for adaptive sampling. To produce a simpler control structure and permit efficient implementation on a mobile GPU platform, however, our adaptive undersampling algorithm adopts a simple traversal mechanism, whereby the pixels are processed in a fixed order: base pixels, type-A1 adaptive pixels, and type-A2 adaptive pixels. In this paper, we describe our algorithm in terms of the 2×2 pixel partitioning shown in Fig. 2. It requires only a simple modification to handle a base block of larger size.

3.2 Stage I: regular sampling of base pixels

In the first stage of our algorithm, a ray is traced recursively through each of the B pixels. Whereas the eventual goal of firing a primary ray for each pixel is to compute the final shaded color (COL), our method collects various ray attributes at the first hit of the ray, which are exploited later to enable efficient rendering computations. These include a set of geometry attributes of the surface at the first hit, comprising an object identification number (OID), a position vector (POS), a normal vector (NORM), shadow bits (SHDBIT), and texture coordinates (TCOORD), where the SHDBIT attribute stores a set of shadow bits such that a bit is set if and only if a shadow is cast at the surface point with respect to the corresponding light source. In addition, a global shaded color (GCOL) attribute is collected. In our current implementation, this stores the radiance from specular reflection and refraction, although any other radiance caused by a different kind of global illumination may be associated with GCOL. In this work, the vector (OID, POS, NORM, SHDBIT, TCOORD, GCOL) is called the ray-attribute vector (or simply attribute vector) for a pixel.

3.3 Stages II and III: adaptive sampling of adaptive pixels

The actual adaptive ray-sampling computation proceeds in two separate steps using the ray-attribute vectors of the B pixels as the initial data. Each elementary sampling operation in stages II and III takes the attribute vectors of two reference pixels as inputs and computes an attribute vector for an adaptive pixel, called the current pixel that exists hor-

Table 1 Similarity checks

	Test conditions
G1	$OID_0 = OID_1$
G2	$\ POS_0 - POS_1\ \leq T_{pos}$
G3	$NORM_0 \circ NORM_1 \geq T_{norm}$
G4	$\{(POS_1 - POS_0) \circ NORM_0\} \cdot \{(POS_0 - POS_1) \circ NORM_1\} \geq 0$
TX	$TCOORD_{\alpha} \cdot \beta \geq T_{tex}$ or $TCOORD_{1-\alpha} \cdot \beta \leq 1.0 - T_{tex}$
SH	$SHDBIT_0 = SHDBIT_1$
GC	$\ GCOL_0 - GCOL_1\ \leq T_{gcol}$

The subscripts 0 and 1, respectively, denote two reference pixels. Here, \circ denotes the inner product of two vectors, α can be either 0 or 1, and β refers to either the s or t texture coordinate

izontally (in stage II) or vertically (in stage III) between the reference pixels. In stage II, an attribute vector of each A1 pixel (the current adaptive pixel) is calculated via interpolation or ray tracing based on the attribute vectors of the two neighboring B pixels (the reference pixels) in the same row. In stage III, the same computation is then repeated vertically, taking each A2 pixel as the current pixel and then calculating its attribute vector using those of the corresponding B or A1 pixels in the same column as the reference pixels. Note that when the attribute vector of a pixel is ready, the final color can easily be produced from it.

3.4 Similarity checks

The key aim of our method is to seek to compute the attribute vectors of adaptive pixels through cheap interpolation, in which the linear interpolation for each ray attribute is clearly defined, as much as possible, instead of through expensive ray tracing. To check if simple linear interpolation may safely be applicable, a series of seven elementary tests called similarity checks are performed (refer to Table 1 for a summary of these tests). In our method, the interpolation is applied only if all the tests succeed.

Four local geometry tests The aim of these four tests is to examine if the four geometry attributes, OID, POS, NORM, and TCOORD, can be interpolated from those of the reference pixels. First, different objects between two pixels are often the most serious source of annoying aliases. Therefore, the first test compares the OIDs of two reference pixels and is considered to fail if the objects are different from each other (G1 in Table 1). Second, the next test checks if the distance between the first hits of the reference pixels is less than a given distance threshold (G2). Third, the normal direction at the first hit is particularly useful for detecting an edge formed by polygons of an object that meet at an acute angle. Therefore, this third test investigates if the dot product of the

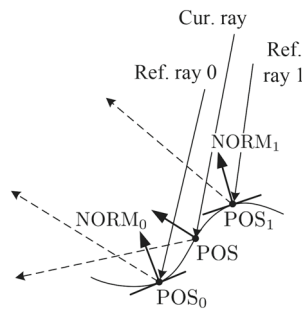


Fig. 3 Convexity check. The local concave geometry around the current ray's first hit may result in annoying alias artifact when the geometry data interpolated from those of the two adjacent reference rays are used for ray tracing

normal directions of the reference pixels is less than a preset threshold (G3).

Although these three similarity tests have often been used in previous methods, they can introduce aliasing when the real intersection point exists on a complex surface. Figure 3 shows a common adverse situation in which the local surface fluctuates between the first hits POS_0 and POS_1 of the reference pixels. In this case, a linear interpolation of the two normal vectors may give an inaccurate normal at the current ray's position POS , even though the previous three tests may have succeeded. An incorrect normal can result in a severe error when the surface point is locally shaded or the reflection/refraction direction is generated.

To minimize these problems with normals, we perform a fourth elementary test, called the convexity check, in which the signs of the first hit point in 3D space with respect to the tangent plane defined by the position and normal at the second first hit, and vice versa, are examined (G4 in Table 1). If the two signs are different, the local surface between POS_0 and POS_1 is not smooth, possibly causing a troublesome fluctuation. Although the success of the convexity check does not guarantee surface convexity, because the surface can have multiple inflections, we have found the convexity check to be quite effective for removing normal-related aliasing.

A shadow test Next, our method performs a shadow test that succeeds only if all the corresponding shadow bits in SHD-BITs of the two reference pixels are identical (SH in Table 1). If the test succeeds, the current pixel simply inherits the light visibility from the reference pixels without shooting shadow rays. Otherwise, if at least one bit field disagrees, the shadow rays are fired toward each light. This all-or-nothing strategy may appear excessive, because the light visibility could be checked only for lights with different visibility. However, this strategy produces a simple control structure that results eventually in more efficient SIMD processing on the mobile GPU platform, particularly for scenes with few lights.

A texture test Often, the same texture image is repeatedly applied to surfaces during texture mapping. If the image is not continuous along its boundaries, a careless linear interpolation of texture coordinates from the two reference pixels could cause annoying aliases. To avoid such problematic situations, we perform a texture test that checks whether, for each component of the texture-coordinate vector, at least one of the corresponding coordinates exists in the interval $[T_{\text{tex}}, 1.0 - T_{\text{tex}}]$ for some small $T_{\text{tex}} > 0$ (TX in Table 1). See Fig. 1f to notice how a single texture image was repeatedly mapped onto the floor surface, where simple interpolation of texture coordinates around the boundaries would easily cause wrong texture fetches.

A global color test The last, but not least, element of classic ray tracing is the effect of indirect illumination caused by specular reflection and refraction, for which costly secondary rays must be traced recursively. In the same way as for primary rays, we may investigate the geometry of these secondary rays via similarity checks, at both the origins and the destinations. However, our preliminary implementation revealed that such a detailed adaptive technique often worsened the runtime performance markedly, at least on the current mobile GPU platform. Therefore, we conduct a simple global color test in which the reflection/refraction colors of the reference pixels are compared with each other (GC in Table 1).

4 Postcorrection of undersampled images

Due to insufficient sampling, our method may introduce the problem that objects, or parts of objects, can fall between ray-traced samples and be missed. Figure 4a, b illustrates a typical situation where the vanishing part in Fig. 4b falls between pixels that are classified, through either ray tracing or interpolation, as being outside the thin object. If those pixels have similar geometry attributes, an incorrect OID is interpolated into the intervening adaptive pixels, making the middle part disappear.

An important observation is that the missing object problem always occurs in the interpolated adaptive pixels that can be traced from ray-traced adaptive pixels. Such troublesome adaptive pixels are marked with an asterisk in Fig. 4b. An effective way of removing such aliasing is to revisit the ray-traced adaptive pixels, marked in thicker lines in Fig. 4b, propagating their correct ray-object intersection information into their interpolated neighbors. Given a ray-traced adaptive pixel x , consider a neighboring pixel y of x , whose geometry attributes have been interpolated. If the OIDs of x and y are different, y becomes a candidate for the problematic pixel, marked with white dots in Fig. 4c. To investigate whether it actually is a candidate, a primary ray is additionally

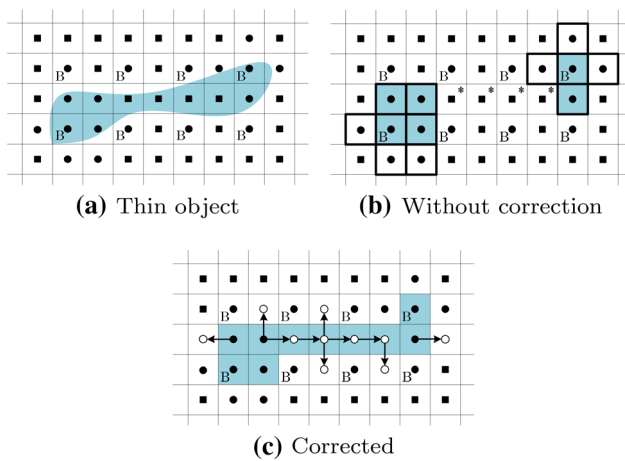


Fig. 4 Correction of missing parts. Here, the *black dots* and the *squares* indicate pixels whose geometry attributes were obtained through regular ray tracing and interpolation, respectively. The base pixels are marked with B. **a** Thin object. **b** Without correction. **c** Corrected

shot through y , thereby performing the regular ray-tracing operation. If the new OID differs from the old one, then a missing part of the object has been found and can be reconstructed. The adaptive pixel y then becomes classified as ray traced, and its interpolated neighbors are repeatedly investigated, as illustrated in Fig. 4c. In this propagation process, an eight-neighbor examination would give a more robust result. However, we find that a more efficient four-neighbor examination produces sufficiently good rendering results for the 2×2 base block.

Notice that our correction algorithm involves postprocessing after the entire pass of adaptive rendering is complete. It differs from a previous approach to pixel-selected ray tracing [1], which aims to detect pixels of vanishing objects by referring to the color attributes of pixels during adaptive ray tracing. By separating the adaptive-sampling and error-correction stages, we can achieve a simpler, GPU-friendly algorithm. Note also that our antialiasing mechanism is selective in that other aliases, such as “missing shadow,” can selectively be reduced by checking the corresponding attribute (e.g., SHDBIT for shadow antialiasing).

5 Efficient implementation on mobile GPUs

Because runtime performance is usually more vulnerable to careless GPU implementation on a current mobile platform than on a PC platform, the GPU program must be carefully tuned for maximum efficiency. First, consider the similarity checks in our seven elementary tests. If all the local geometry tests succeed but the shadow test fails, for instance, we may interpolate the local geometry but shoot shadow rays for light visibility. However, to avoid the branch divergences that have a significant negative impact on the GPU performance, our

Table 2 Seven-kernel implementation of adaptive undersampling

Kernel	Operations
Step-I	<ul style="list-style-type: none"> – Trace a ray for each B pixel and shade it – Store the ray attribute vectors of the B pixels in global memory
Step-II-a	<ul style="list-style-type: none"> – Perform the similarity checks for each A1 pixel – If they pass, interpolate the attribute vector and shade the pixel – If not, store the address of the current pixel in global memory
Step-II-b	– Pack the A1 pixels to be ray traced through parallel scans
Step-II-c	Same as step-I except that the packed A1 pixels are processed
Step-III-a	Same as step-II-a except that the A2 pixels are processed
Step-III-b	Same as step-II-b except that the A2 pixels are processed
Step-III-c	– Trace a ray for each packed A2 pixel and shade it

implementation adopts the strategy of full ray tracing for the current pixel if there is any failure in the similarity checks.

Second, as a result of the similarity checks in stage I and stage II, a set of usually sparse problematic pixels are detected, for which expensive ray tracing is to be carried out in the next stage. Again, to minimize the branch divergence between concurrent threads, our implementation runs a separate kernel for packing those pixels into a contiguous region before initiating the ray-tracing computation. Despite this extra kernel requiring a series of parallel scan operations [17] on the GPU, our test results exhibit a significant enhancement in the rendering performance, because the pack operation also reduces global memory bandwidth significantly. Table 2 summarizes our seven-kernel implementation of the proposed algorithm, which shows the highest rendering performance. Note that splitting the GPU program into kernels of smaller granularity might improve the GPU efficiency further. However, we observe that the increasing numbers of global memory accesses cancels out the benefit from the reduced divergence, ultimately reducing the GPU efficiency.

Third, while the optional postcorrection technique in Sect. 4 can easily be implemented using a stack on a CPU, a different implementation scheme is needed for effective many-core processing. In our method, a concurrent thread, associated with each row of the image, first scans its row from left to right, detecting and correcting problematic pixels progressively. The same operation is then carried out repeatedly from right to left, from top to bottom, and finally from bottom to top. This requires four applications of the scanning process, but our experiments have also shown that scanning in just two orthogonal directions, e.g., from left to right and from top to bottom, usually produces sufficiently good correction outcomes.

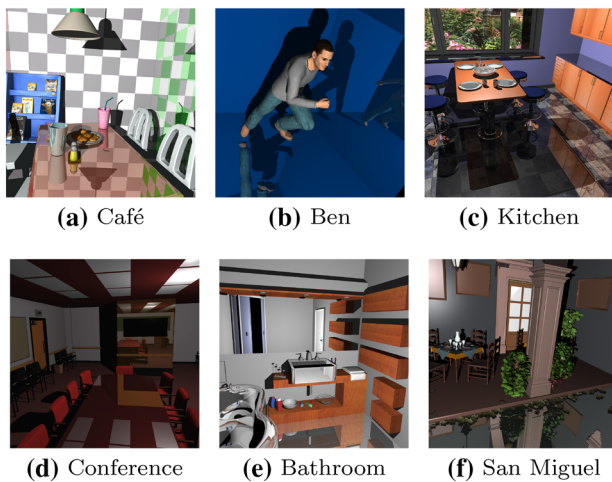


Fig. 5 Example scenes and the camera views tested. To achieve fair evaluation of our method on a mobile phone, we selected six scenes with low to high geometric and rendering complexity, whose triangle numbers ranged from 29,359 to 588,402. Because of the limited memory space of the tested mobile phone, some part of the original dataset for San Miguel was omitted. Note that, because the distance threshold T_{pos} for the local geometry test G2 is dependent on the dimension of the scene, each scene was normalized such that the longest side of the axis aligned bounding box has length 1. **a** Café. **b** Ben. **c** Kitchen. **d** Conference. **e** Bathroom. **f** San Miguel

6 Experimental results

To test our method, we first implemented a *kd-tree*-based full ray tracer using the OpenCL 1.2 API on an LG G3 Cat.6 mobile phone that uses the Qualcomm Snapdragon 805 chipset equipped with an Adreno 420 GPU. The proposed adaptive undersampling technique was then applied to optimize the rendering computation on the mobile platform. All the timings were measured using OpenCL workgroups of 8×8 work items and default thresholds $T_{\text{pos}} = 0.03$, $T_{\text{norm}} = 0.9$, $T_{\text{tex}} = 0.3$, and $T_{\text{gcol}} = 0.15$, which generally produced good results.

6.1 Computation time

Table 5 at the end of this article compares our method to full ray tracing, which shoots one ray through every pixel (see the sampling—1 \times 1 rows). The timing results in the Time and Speedup columns indicate that the proposed adaptive sampling method (Ours) compares quite favorably to the nonadaptive method (Full RT), being 1.48–2.21 times faster when the 2×2 base block was used to render 1024×1024 images for the six example scenes shown in Fig. 5. This efficiency gain was achieved primarily by the decrease in the costly ray-tracing computation despite the extra overhead for the adaptive undersampling, where the figures in the RT ratio column show that only 27.0–34.5 % of image pixels, including both base and adaptive pixels, were actually ray

traced in our method. Figure 1 shows those adaptive pixels that were found to be problematic in the respective similarity checks.

Despite our efforts toward lowering the ray-tracing cost, it still accounts for a major portion of the rendering computation, which paradoxically shows the importance of adaptive undersampling on the mobile GPU. As implied by the timing results in Table 3a, which reports the breakdown of runtimes for the test scenes measured for 1024×1024 images, the kernel *step-I*, *step-II-c*, and *step-III-c* spent 64.4 % (Conference) to 78.3 % (Kitchen) of the rendering time to ray trace around 30 % of 1024^2 pixels, whereas the other kernels, including the optional postcorrection (*step-IV*) and the overhead of initiating the GPU program and transferring data (*step-ETC*), used the remaining time to shade the other pixels.

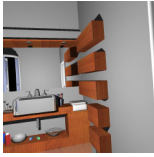



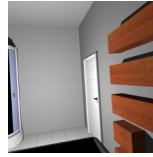
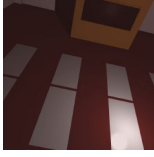
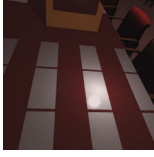
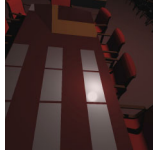

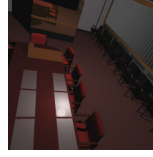
Note that the base pixels comprising one-quarter of the entire image pixels are always ray traced when the 2×2 base block is employed, implying that the lowest possible ray-tracing ratio is 25 %. With increasing scene and rendering complexity, the ratio will increase to maintain the rendering quality, in turn lowering the speedup number. Otherwise, there would be an incorrect reliance on heavy interpolation. When the complexity is beyond the capability of a given ray-sampling density, spatial aliasing artifacts occur even for full ray tracing (the case when the ratio is 100 %), to which supersampling has been an inevitable solution. The statistics in the sampling— $2 \times 2/4 \times 4$ rows in Table 5 show that the ray-tracing pixel ratio decreases, thereby improving the efficiency in the supersampling settings.

6.2 Image quality

Figure 6a–c depicts an example of how effectively the postcorrection algorithm reconstructs the vanishing parts of thin objects and the shadow cast by them in the bathroom scene, where our basic adaptive undersampling method suffered from the “missing object” problem. At the extra cost of postcorrection through the OID and SHDBIT attributes, our method was able to reconstruct these missing parts, resulting in an image that appeared very similar to that produced via full ray tracing. Note that, for the tested scenes, the correction stage required 7.7 % (Ben) to 17.1 % (Conference) of the entire rendering time, resulting in a slight decrease in the frame rate. Although the timings in Table 5 include that for the postcorrection computation, this feature can often be turned off for such scenes as café, Ben, and Conference that do not contain very thin objects, which would produce an additional performance enhancement without significant harm to the rendering quality.

However, the postcorrection algorithm could not effectively remove the other kind of aliasing artifact, such as that appearing on the sink surface where the floor surface,

Table 3 Analysis of the timing performance of our adaptive undersampling method

	I	II-a	II-b	II-c	III-a	III-b	III - c	IV	ETC
(a)									
Café	284.3	25.9	10.4	53.7	38.5	11.1	94.6	60.8	11.8
Ben	329.8	25.4	11.5	40.2	36.8	10.7	66.7	44.3	9.8
Kitchen	341.7	26.3	10.2	99.4	37.5	10.7	207.1	81.9	13.2
Conference	206.4	23.8	10.0	38.7	37.3	11.6	81.8	86.7	11.6
Bathroom	300.7	27.0	10.0	84.9	37.1	11.1	185.2	66.5	11.0
San Miguel	362.7	24.0	10.7	126.0	34.3	11.0	198.1	130.0	8.7
(b)									
Ratio of reflective pixels	19.9 %	41.1 %		60.9 %		82.0 %		99.9 %	
Full ray tracing (ms)	813.3	870.2		992.8		1054.4		1077.3	
Ours (ms)	454.0 (1.79×)	451.8 (1.92×)		495.1 (2.00×)		493.5 (2.13×)		437.0 (2.46×)	
Camera view tested									
(c)									
Ratio of ray traced pixels	25.9 %	26.6 %		28.0 %		29.1 %		32.1 %	
Full ray tracing (ms)	944.6	884.7		866.4		859.8		904.7	
Ours (ms)	362.1 (2.60×)	414.4 (2.13×)		460.4 (1.88×)		521.7 (1.64×)		606.1 (1.49×)	
Camera view tested									

(a) The dissection of the kernel execution time (ms). The times for the kernels **step-I**, **step-II-c**, and **step-III-c**, involving the ray-tracing operation, still account for a large proportion of the rendering time despite our efforts to reduce it, which paradoxically shows the importance of adaptive undersampling. Here, the postcorrection is performed in the final kernel **step-IV**

(b) The speedups with respect to the ratio of image pixels for which reflection rays are fired (bathroom). Unlike full ray tracing, whose rendering time increased rapidly as the reflection rays increased, our method was less sensitive to these extra rays because the ray-tracing computation was suppressed effectively

(c) The speedups with respect to the ratio of image pixels for which the ray-tracing computation is performed (Conference). This ratio, which is always larger than 25 % in the current scheme, is one of the most influential factors affecting the timing efficiency of our method, where it usually fell within the given range

All test scenes were rendered at 1024×1024 pixels using single sampling

represented as a single object, is reflected (see Fig. 6d, e). These artifacts are caused mainly by the small details on the reflected floor surface being simply beyond the capability of the applied sampling density of one sample per pixel (sampling— 1×1). They also appear in the full ray-traced image. Although the adaptive undersampling technique worsens the situation somewhat for temporal efficiency reasons, the appropriate solution is again supersampling, where the 16 ray samplings per pixel (sampling— 4×4) reduce the visual difference between the results for our method and for full ray tracing (see Fig. 6f, g).

Overall, our experiments show that good image quality is maintained despite the reduced numbers of ray shots, as given in the PSNR column in Table 5. Figure 6h, i also compares the results from the full ray tracing (top) and our adaptive undersampling (middle) for two example scenes. The most obvious visual errors, as displayed in the difference image (bottom), usually occur around corners or for highly curved objects, which are often hard to detect using similarity checks. Furthermore, when textures are applied, the shaded colors of interpolated adaptive pixels differ slightly from those of the ray-traced pixels. However, these visual differences are often difficult to detect, particularly when rendered interactively.

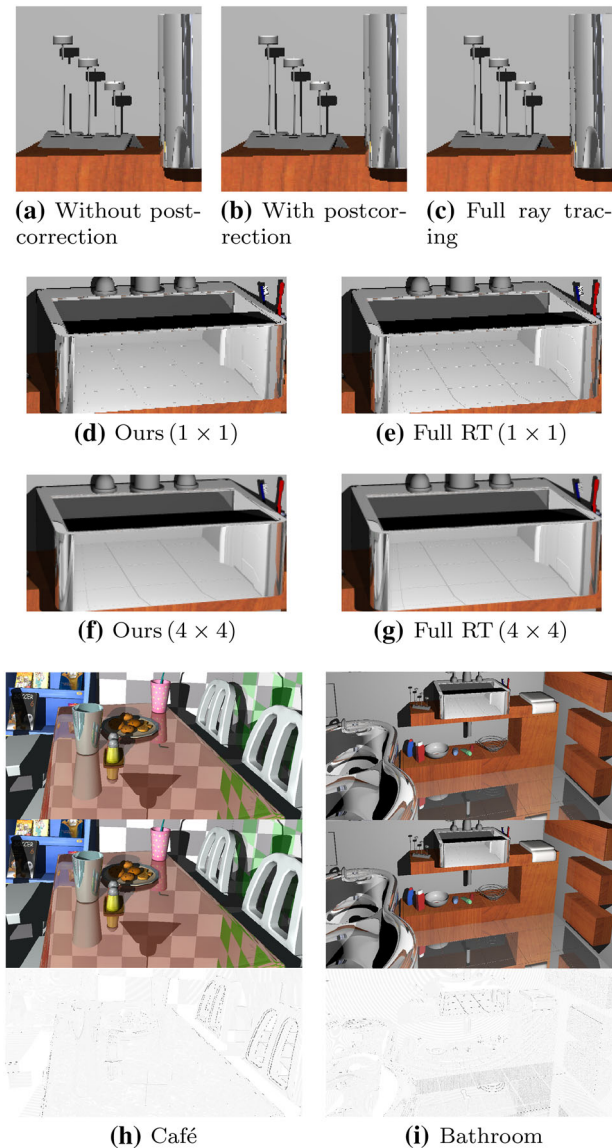


Fig. 6 Comparison of rendering results. Parts of the images, which were rendered at 1024×1024 pixels, are shown to aid analysis of the rendering quality. See the text for detail. **a** Without postcorrection. **b** With postcorrection. **c** Full ray tracing. **d** Ours (1×1). **e** Full RT (1×1). **f** Ours (4×4). **g** Full RT (4×4). **h** Café. **i** Bathroom

6.3 Further analysis

Our method is usually more effective when more shadow and/or reflection/refraction rays are to be traced. If full ray tracing is employed, the extra rendering cost increases linearly with the additional number of these rays. As clearly indicated in the experiment where the ratio of pixels for which reflective objects are visible (i.e., reflection rays are being fired), is varied (see Table 3b), the rendering cost for handling the extra secondary rays increased slowly in our method because the ray-tracing computation was suppressed effectively.

In summary, the timing performance of our method was primarily affected by the ratio of pixels, including base pixels, for which ray tracing should be performed. This is clearly confirmed in the experiment where the ratio was varied in the interval that usually contains those observed in the tested example scenes (refer to Table 3c). Note that our method is selectively controllable in that the performance drop can be suppressed by lowering this ratio through relaxed tolerances in the relevant similarity checks. Developing an effective way to find an optimal set of tolerance values for input scenes remains an open problem.

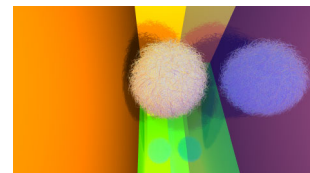
7 Concluding remarks

As noted earlier, many-core processing driven with a parallel programming tool such as the OpenCL API is more vulnerable to the complexity of parallel algorithms on mobile GPUs than on PC-based GPUs. Therefore, it was critical to design a mobile GPU algorithm with a simple control structure, which explains why we had to compromise between simplicity and flexibility in developing our algorithm. As a result, our adaptive sampling scheme is orthogonal to the acceleration structures and traversal algorithms that are routinely used in ray tracing. We expect that it will be combined effectively with a mobile ray-tracing hardware architecture for even higher ray-tracing throughput in the future.

In the future, it will be worthwhile to investigate the possibility of our method in a PC GPU ray tracer. Our preliminary experiments show that the simple porting of the OpenCL-based ray tracer also allows effective adaptive ray sampling on a PC platform when high-quality, high-resolution images are to be ray traced for complicated scenes. Table 4 summarizes the statistics collected when a reflective Hairball model made of 2,880,000 triangles was rendered at the 4K UHD res-

Table 4 Preliminary performance test on a PC platform

Sampling	Ours (time: s, PSNR: dB)				
	FRT Time	Time	Speedup	RT ratio	PSNR
1×1	5.2	2.6	2.00×	0.390	31.72
2×2	18.9	9.4	2.01×	0.378	37.73
4×4	66.6	32.6	2.04×	0.360	42.21
8×8	232.1	110.3	2.10×	0.343	45.62
16×16	803.8	369.6	2.17×	0.329	48.16



Tested view

Three lights were lit to ray trace the reflective Hairball model where the floor and the right wall were also reflective

Table 5 Performance comparison with full ray tracing (2×2 base block). Six scenes of low to high geometric complexity were tested, with triangle numbers given in parentheses. The figures in the RT ratio column indicate the ratios of the number of ray samples used by our renderer (“ours”) to that for the full ray tracer (“full RT”). The PSNR values were measured by comparing the respective rendering images produced by the two methods

Scene	Resolution	Sampling	Full RT	Ours	Speedup	RT ratio	PSNR (dB)
			Time (ms)	Time (ms)			
Café (29,359)	512 ²	1 × 1	314.0	199.3	1.57×	0.327	43.44
		2 × 2	1203.1	637.3	1.88×	0.294	41.72
		4 × 4	4620.6	2130.1	2.16×	0.274	41.68
	1024 ²	1 × 1	1211.7	591.2	2.04×	0.294	43.99
		2 × 2	4396.7	2079.5	2.11×	0.274	44.03
		4 × 4	16, 663.8	7382.3	2.25×	0.263	44.26
	2048 ²	1 × 1	4332.0	1928.5	2.24×	0.274	44.58
		2 × 2	16, 593.7	6991.6	2.37×	0.263	46.23
		4 × 4	83, 840.8	25, 808.9	3.24×	0.257	46.58
Ben (78,039)	512 ²	1 × 1	345.2	191.7	1.79×	0.288	44.25
		2 × 2	1314.3	619.3	2.12×	0.270	47.81
		4 × 4	4802.3	2086.2	2.30×	0.260	50.92
	1024 ²	1 × 1	1275.8	575.2	2.21×	0.270	45.46
		2 × 2	4586.9	1946.6	2.35×	0.260	49.55
		4 × 4	18, 007.2	7230.0	2.49×	0.255	51.66
	2048 ²	1 × 1	4654.1	1927.7	2.41×	0.260	47.23
		2 × 2	17, 789.7	6868.0	2.59×	0.255	51.00
		4 × 4	78, 791.9	26, 620.5	2.95×	0.253	52.14
Kitchen (101,015)	512 ²	1 × 1	356.0	278.8	1.27×	0.394	37.89
		2 × 2	1319.8	834.0	1.58×	0.345	43.99
		4 × 4	4870.8	2866.7	1.69×	0.312	48.40
	1024 ²	1 × 1	1296.9	828.0	1.56×	0.345	39.14
		2 × 2	4796.9	2656.5	1.80×	0.312	45.22
		4 × 4	18, 244.1	8978.5	2.03×	0.291	48.83
	2048 ²	1 × 1	4777.8	2634.1	1.81×	0.312	39.79
		2 × 2	17, 738.9	8620.8	2.05×	0.291	46.04
		4 × 4	68, 734.2	29, 885.4	2.29×	0.276	49.14
Conference (190,947)	512 ²	1 × 1	230.6	173.1	1.33×	0.336	47.44
		2 × 2	842.5	549.0	1.53×	0.300	50.56
		4 × 4	3414.4	1958.2	1.74×	0.277	51.11
	1024 ²	1 × 1	841.6	508.1	1.65×	0.300	48.92
		2 × 2	3284.0	1803.9	1.82×	0.277	50.35
		4 × 4	12, 700.4	6729.4	1.88×	0.265	51.58
	2048 ²	1 × 1	3216.5	1666.2	1.93×	0.277	50.89
		2 × 2	12, 640.0	6157.8	2.05×	0.265	51.41
		4 × 4	49, 223.0	23200.3	2.12×	0.258	51.61
Bathroom (268,725)	512 ²	1 × 1	319.4	240.6	1.32×	0.355	36.63
		2 × 2	1223.8	857.0	1.42×	0.318	44.31
		4 × 4	4384.2	2805.8	1.56×	0.294	48.60
	1024 ²	1 × 1	1169.5	733.6	1.59×	0.318	40.54
		2 × 2	4331.8	2561.0	1.69×	0.294	46.41
		4 × 4	16, 759.8	8480.6	1.97×	0.280	49.12
	2048 ²	1 × 1	4293.7	2456.2	1.74×	0.294	41.85

Table 5 continued

Scene	Resolution	Sampling	Full RT	Ours			
			Time (ms)	Time (ms)	Speedup	RT ratio	PSNR (dB)
San Miguel (588,402)	512 ²	2 × 2	16,555.3	8183.9	2.02×	0.280	47.25
		4 × 4	64,030.8	28,831.4	2.22×	0.270	49.85
		1 × 1	382.5	324.2	1.17×	0.395	36.87
		2 × 2	1357.0	978.4	1.38×	0.340	43.10
		4 × 4	4910.6	3375.9	1.45×	0.300	46.47
		1024 ²	1 × 1	1344.3	905.4	1.48×	0.340
	2048 ²	2 × 2	4833.8	2923.6	1.65×	0.300	43.18
		4 × 4	17,917.2	9,987.0	1.79×	0.278	48.08
		1 × 1	4814.2	2838.4	1.69×	0.300	38.06
		2 × 2	17,845.4	8774.0	2.03×	0.278	45.89
		4 × 4	67,949.8	31,442.0	2.16×	0.265	49.51

olution of 3840×2160 pixels on a desktop PC with an AMD Radeon R9 Fury X GPU. Here, because of the complexity of the model, high rates of sampling was needed to ensure the rendering quality. As in the mobile ray tracing, we observe that our method achieves marked speedups while retaining good image quality compared to the full ray tracing. Tailoring our adaptive undersampling algorithm to best fit the PC GPU remains a future research topic.

Acknowledgments The test scenes are courtesy of I. Wald (Ben), J. Helenklaken (Kitchen), A. Grynberg and G. Ward (Conference), G. M. Leal Llaguno (San Miguel), and S. Laine and T. Karras (Hairball). This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. NRF-2015R1A2A2A01006590).

References

- Akimoto, T., Mase, K., Suenaga, Y.: Improved pixel selected ray tracing. *Syst. Comput. Jpn.* **22**(4), 57–67 (1991)
- Cook, R.: Stochastic sampling in computer graphics. *ACM Trans. Graph.* **5**(1), 51–72 (1986)
- Dippé, M., Wold, E.: Antialiasing through stochastic sampling. In: *Proceedings of SIGGRAPH 1985*, pp. 69–78 (1985)
- Formella, A., Gill, C., Hofmeyer, V.: Fast ray tracing of sequences by ray history evaluation. In: *Proceedings of computer animation 1994*, pp. 184–191 (1994)
- Genetti, J., Gordon, D., Williams, G.: Adaptive supersampling in object space using pyramidal rays. *Comput. Graph. Forum* **17**(1), 29–54 (1998)
- Hachisuka, T., Jarosz, W., Weistroffer, R., Dale, K., Humphreys, G., Zwicker, M., Jensen, H.: Multidimensional adaptive sampling and reconstruction for ray tracing. *ACM Trans. Graph. (Proc. ACM SIGGRAPH)* **27**(3), Article No. 33 (2008)
- He, Y., Gu, Y., Fatahalian, K.: Extending the graphics pipeline with adaptive, multi-rate shading. *ACM Trans. Graph. (Proc. ACM SIGGRAPH)* **33**(4), Article No. 142 (2014)
- Jin, B., Ihm, I., Chang, B., Park, C., Lee, W., Jung, S.: Selective and adaptive supersampling for real-time ray tracing. In: *Proceedings of the ACM conference on high performance graphics (HPG 2009)*, pp. 117–125 (2009)
- Kajiya, J.: The rendering equation. In: *Proceedings of SIGGRAPH 1986*, pp. 143–150 (1986)
- Lee, M., Redner, R., Uselton, S.: Statistically optimized sampling for distributed ray tracing. In: *Proceedings of SIGGRAPH 1985*, pp. 61–67 (1985)
- Levoy, M.: Volume rendering by adaptive refinement. *Vis. Comput.* **6**(1), 2–7 (1990)
- Mitchell, D.: Generating antialiased images at low sampling densities. In: *Proceedings of SIGGRAPH 1987*, pp. 65–72 (1987)
- Murakami, K., Hirota, K.: Incremental ray tracing. In: *Proceedings of eurographics workshop on photosimulation, realism and physics in computer graphics 1990*, pp. 15–29 (1990)
- Ohta, M., Maekawa, M.: Ray-bound tracing for perfect and efficient anti-aliasing. *Vis. Comput.* **6**(3), 125–133 (1990)
- Painter, J., Sloan, K.: Antialiased ray tracing by adaptive progressive refinement. *Comput. Graph. (ACM SIGGRAPH 1989)* **23**(3), 281–288 (1989)
- Rigau, J., Feixas, M., Sbert, M.: Refinement criteria based on f -divergences. In: *Proceedings of eurographics symposium on rendering 2003*, pp. 260–318 (2003)
- Sengupta, S., Harris, M., Garland, M., Owens, J.D.: Efficient parallel scan algorithms for many-core GPUs. In: *Scientific Computing with Multicore and Accelerators*, chap. 19, pp. 413–442. Taylor & Francis, Abingdon (2011)
- Thomas, D., Netravali, A., Fox, D.: Antialiased ray tracing with covers. *Comput. Graph. Forum* **8**(4), 325–336 (1989)
- Vaidyanathan, K., Salvi, M., Toth, R., Foley, T., Akenine-Moller, T., Nilsson, J., Munkberg, J., Hasselgren, J., Sugihara, M., Clarberg, P., Janczak, T., Lefohn, A.: Coarse pixel shading. In: *Proceedings of the ACM SIGGRAPH symposium on high performance graphics 2014*, pp. 9–18 (2014)
- Whitted, T.: An improved model for shaded display. *Commun. ACM* **23**(6), 343–349 (1980)



Youngwook Kim received his BSE degree in computer engineering from Hankuk University of Foreign Studies, Korea in 2012. He is currently a doctoral student in computer science and engineering at Sogang University, Korea. His research interests include real-time rendering and mobile computing.



Yeongkyu Lim received his BS degree from Kyungpook National University, Korea in 1997 and received his MS degree from Department of Computer Science at Korea University in 1999. In 2013, he received his Ph.D. degree from Department of Computer Science at Yonsei University, Korea. He has worked on LG Electronics since 1999. His research areas are embedded systems, HCI and mobile GPU architecture and computing.



Woong Seo received his BSE degree in electrical engineering from Hankuk University of Foreign Studies, Korea in 2012. He received his MSE degree in computer science and engineering from Sogang University in 2014, where he is currently pursuing a Ph.D. degree. His research interests include real-time rendering and mobile distributed rendering.



Jae-Ho Nah received his BS, MS, and Ph.D. degrees from the Department of Computer Science, Yonsei University, Korea in 2005, 2007, and 2012, respectively. Currently, he is a senior research engineer at LG Electronics. His research interests include ray tracing, rendering algorithms, and graphics hardware.



Yongho Kim received his BSE and MSE degrees in computer science and engineering from Sogang University, Korea in 2013 and 2015, respectively. He is currently developing various games at NCSOFT. His research interests include computer graphics, 3D games, and virtual reality.



Insung Ihm received his BS degree in computer science and statistics from Seoul National University, Korea in 1985 and his MS degree in computer science from Rutgers University, USA in 1987. Then, he received his Ph.D. degree in computer science from Purdue University, USA in 1991. He is currently a professor in computer science and engineering at Sogang University, Korea. His research interests include computer graphics, scientific visualization, and high-performance computing.