Selective and Adaptive Supersampling for Real-Time Ray Tracing

Bongjun Jin Insung Ihm Byungjoon Chang
Department of Computer Science and Engineering
Sogang University, Korea

Chanmin Park Wonjong Lee Seokyoon Jung
System Architecture Laboratory
Samsung Electronics, Korea

Abstract

While supersampling is an essential element for high quality rendering, high sampling rates, routinely employed in offline rendering, are still considered quite burdensome for real-time ray tracing. In this paper, we propose a selective and adaptive supersampling technique aimed at the development of a real-time ray tracer on today's many-core processors. For efficient utilization of very precious computing time, this technique explores both image–space and object–space attributes, which can be easily gathered during the ray tracing computation, minimizing rendering artifacts by cleverly distributing ray samples to rendering elements according to priorities that are selectively set by a user. Our implementation on the current GPU demonstrates that the presented algorithm makes high sampling rates as effective as 9 to 16 samples per pixel more affordable than before for real-time ray tracing.

CR Categories: I.3.1 [Computer Graphics]: Hardware architecture—Graphics processors; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

Keywords: Real-time ray tracing, adaptive supersampling, selective sampling, many-core processor, GPU computing.

1 Introduction

1.1 Background and our contribution

Annoying alias artifacts often occur in ray tracing for a variety of reasons because its computation inherently involves several rendering elements such as ray–object intersection, reflection/refraction, shadows, textures, etc. One naïve method for removing or, more exactly, reducing aliases, is to take more samples over each pixel and combine the resulting colors using a proper filter. In offline rendering, it is not uncommon to take, for example, 16 samples per pixel to generate high quality renderings. However, such a high sampling rate is still too heavy for real-time ray tracing when the sampling process is implemented recklessly. A clever approach is to optimize rendering computation by adaptively firing rays only where necessary. Although some hybrid methods have been proposed before, image-space techniques have usually been used for adaptive supersampling, in which a color-related measure alone is applied in an adaptive manner to detect *problematic* regions.

While simple and widely used, such image-space techniques have an inevitable problem in that, whichever criterion is used, it is difficult to effectively locate all artifacts that arise in a complex manner from various sources and are hidden once rays are finally shaded. Generally speaking, the image–space methods are *not selective* in the point that identical criteria are applied to entire image pixels regardless of their content in object space. Because time is still very limited in real-time rendering, it is desirable that users be capable of focusing computing effort on selectively chosen rendering features, improving the most needed ones as much as time allows.

Another notable aspect of previous adaptive sampling techniques is that their computation entails *data-dependent, unpredictable control flows* as sampling proceeds progressively to detect problematic regions based on on-the-fly color comparison. Although this is, in fact, the very property that leads to efficient sampling, it could deteriorate rendering performance when adaptive sampling is implemented on today's many-core processors such as GPU, which favors a simple control structure for extreme performance. To fully harness the massively parallel computing power offered by the current GPU, it is necessary to design a simple adaptive sampling mechanism well suited for its computing architecture.

In this paper, we present a selective and adaptive supersampling technique that is aimed at real-time ray tracing on many-core processors. In addition to image–space color measure, our sampling algorithm explores geometry attributes such as object identification number, object normal, shadow and texture existence that can be easily collected during ray tracing. By adjusting a multivalued threshold vector, the user can selectively set priorities for various rendering features, so that more ray samples can be used to reduce artifacts related to the features with higher priorities.

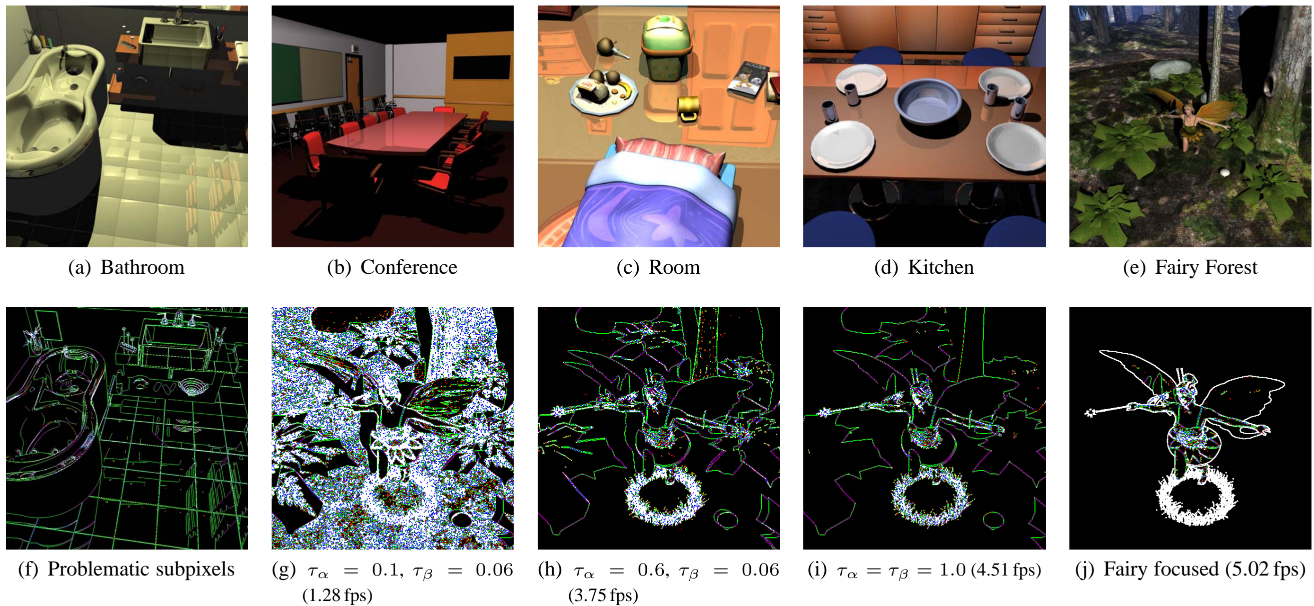
The proposed algorithm is simple in structure and easily mapped to the current GPU architecture, offering an efficient parallel supersampling computation. Despite being originally designed for the GPU, the presented technique is also equally extendable to the current multicore CPU with SIMD capability. More importantly, it enables users to maximize the antialiasing effect by selectively allocating more of the very limited computation time to possibly more troublesome rendering features. We demonstrate that by applying our selective and adaptive supersampling technique, high sampling rates that are as effective as 9 to 16 samples per pixel can become more affordable for real-time ray tracing.

2 Related work

Since first discussed by Crow [Crow 1977] in the middle of 1970s, aliasing has been a major problem in developing a ray tracer. In order to produce high quality antialiased images at reasonable sample rates, Whitted suggested adaptive supersampling in which pixels were recursively subdivided for further sampling only when colors sampled at their four corners varied significantly [Whitted 1980]. In the development of the distributed ray tracing algorithm, Cook et al. proposed to use stochastic sampling that replaced highly objectionable artifacts by noise that is less conspicuous to viewers [Cook et al. 1984]. Mitchell presented effective nonuniform sampling patterns and applied a contrast measure in an aim to reduce 'visible' artifacts where red, green, and blue contrasts were approximated and compared against separate thresholds [Mitchell 1987]. Painter and Sloan presented hierarchical adaptive stochastic sampling for ray tracing that worked in progressive manner. While these previous works were proven to be effective, they often failed to detect small details because they used no information other than sampled inten-

Copyright © 2009 by the Association for Computing Machinery, Inc.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions Dept, ACM Inc., fax +1 (212) 869-0481 or e-mail permissions@acm.org.
HPG 2009, New Orleans, Louisiana, August 1 – 3, 2009.
© 2009 ACM 978-1-60558-603-8/09/0008 $10.00

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org



(a) Bathroom (b) Conference (c) Room (d) Kitchen (e) Fairy Forest

(f) Problematic subpixels (g) $\tau_\alpha = 0.1$, $\tau_\beta = 0.06$ (1.28 fps) (h) $\tau_\alpha = 0.6$, $\tau_\beta = 0.06$ (3.75 fps) (i) $\tau_\alpha = \tau_\beta = 1.0$ (4.51 fps) (j) Fairy focused (5.02 fps)

Figure 1: *Selective and adaptive supersampling. Images in (a) to (e) show five tested scenes rendered at* 1024×1024*. In images in (f) to (j), the number of subpixels in the pixel that were sampled with 4 samples are color-coded: black (0), red (1), green (2), blue (3), and white (4). To achieve a high PSNR value for Fairy Forest that contains highly detailed textures, strict thresholds had to be used ((g)). Using our selective scheme, however, we could adjust them so that more ray samples were directed to perceptually more visible artifacts where the thresholds in (h) offered a similar visual quality as a nonadaptive sampling done with 9 samples per pixel. In situations when the viewer's attention was focused on the fairy, we could further lower the criteria without causing much trouble ((i) & (j)). Image in (e) was created with the thresholds in (i). Note that* τ_α $(= \tau_{col}, \tau_{pte}, \tau_{ste})$ *and* τ_β $(= \tau_{psc}, \tau_{ssc})$ *are useful for controlling aliases from textures and shadows, respectively.*

sities. In order to overcome the aliasing problems resulting from the point sampling approach of ray tracing, beam and cone tracing techniques were proposed respectively by Heckbert and Hanrahan [Heckbert and Hanrahan 1984], and Amanatides [Amanatides 1984], which considered fractional coverage information inside image pixels. Object space information has also been exploited by Thomas et al. [Thomas et al. 1989] and Ohta and Maekawa [Ohta and Maekawa 1990]. Whitted's adaptive sampling scheme [Whitted 1980] was also extended by Genetti et al. in a way that decisions regarding extra sampling were made based on object-space information obtained during the ray-object intersection computation [Genetti et al. 1998]. Perceptually based approach has also been proposed for adaptive sampling [Bolin and Meyer 1998]. Recently, Longhurst et al. utilized the GPU to apply Laplacian filter to find jaggies in rendering images obtained by an OpenGL based renderer, and passed the detected edge information to ray tracer for adaptive supersampling [Longhurst et al. 2005].

In the last few years, real-time ray tracing has become a reality thanks to the increasing power of modern processors. Recently, several researchers have presented various efficient implementation techniques, optimized for respective multicore and many-core processors, for instance, [Wald 2004; Foley and Sugerman 2005; Wald et al. 2007a; Horn et al. 2007; Popov et al. 2007; Shevtsov et al. 2007; Wald et al. 2007b; Zhou et al. 2008; Wald et al. 2008; Overbeck et al. 2008]. A real-time ray tracer was also developed successfully on Intel's upcoming Larrabee processor [Seiler et al. 2008]. So far, fast construction and efficient traversal of spatial data structures for static and dynamic scenes have been the most actively researched topics in the field of real-time ray tracing. On the other hand, the aliasing problem that is caused by the inevitable point sampling of ray tracing has attracted little concern in spite of its importance. As cited in the above paragraph, most antialiasing techniques suitable for ray tracing were developed in the 1980s and 1990s. While effective, they, in their current forms, are not best suited for effective implementation on the computing architecture of today's many-core processors, an example of which will be briefed in the next section.

3 GPU architecture model

In this work, we assume the following parallel programming model specified by the CUDA (Compute Unified Device Architecture) API suitable for developing applications on highly parallel, multithreaded, many-core processors [NVIDIA 2008]. In this model, a processing system consists of a scalable array of multiprocessors on which blocks of threads are run in parallel. Each multiprocessor containing a set of streaming processors have per-thread *registers* and per-block on-chip *shared memory* that allow very high speed read/write access in a highly parallel manner. The shared memory is visible to all threads in the same block and provides an efficient means of cooperation between them. One of the fundamental keys for achieving high performance in this computing model is to *launch a kernel with a massive number of concurrent threads* so that latencies due to off-chip memory access and synchronization are hidden via thread swapping. However, as the registers and shared memory should be partitioned among threads of all resident thread blocks, the number of executable threads in parallel on a multiprocessor is limited by the amount of shared resources that each thread demands.

All threads, whether in the same block or not, can have read/write access to the same global memory space that is located in DRAM. While large but uncached, this off-chip memory usually requires hundreds of cycles of memory latency per access. Kernel variables that do not fit into registers must be allocated to slow, uncached *local memory* that also resides in DRAM. To avoid memory stalls,

it is important to *fully leverage the fast registers and shared memory* and minimize use of these expensive off-chip memories. There also exist two read-only off-chip memories that are readable by all threads. The first is *constant memory*, which often offers faster and more parallel data access than the global memory. The second is *texture memory*, to which threads can have access in different addressing modes. The hardware texture unit also allows fetching data in various scalar or vector data formats, which can have several performance benefits when some specific data structures are used in a kernel program. Because both memories are cached, substantial performance improvement can be achieved by *ensuring spatial and temporal locality in data access*.

Shared and global memories provide an effective means of communications between concurrent threads. In addition, they can synchronize through a lightweight synchronization barrier. When a kernel is launched, a multiprocessor executes assigned thread blocks in groups of 32 concurrent threads, called *warps*, in SIMT (Single-Instruction Multiple-Thread) fashion. Individual threads in a warp execute one common instruction at a time, but are free to branch independently. When control flow diverges in a warp due to a conditional branch, the warp serializes each branch path taken that could possibly lead to performance loss. From the perspective of kernel programming, designing an algorithm with *a simple control structure that minimizes unpredictable, data-dependent branches* can enhance the performance markedly.

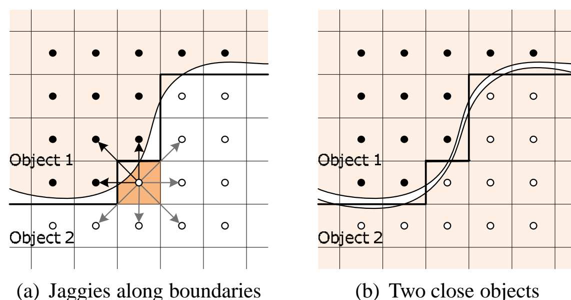
4 Algorithm for selective and adaptive supersampling

In this section, we first explain the basic idea of our method using a simple example that considers only aliases due to undersampling in the intersection computation between the primary ray and objects. Then, a detailed description of our algorithm and its GPU implementation follows.

4.1 Basic idea

4.1.1 Subdivision of a pixel into four subpixels

Figure 2(a) illustrates a situation where a part of an image plane is being sampled by tracing one ray per pixel through its center, creating jagged edges along the silhouette curve formed by Object 1. In our work, it is assumed that such silhouette curves, that is, the objects' projected boundaries, are sufficiently smooth in the screen space in that they do not fluctuate excessively within a pixel area.

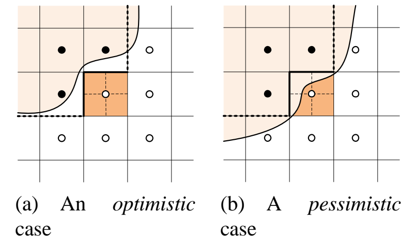


(a) Jaggies along boundaries (b) Two close objects

Figure 2: *Locating problematic pixels. (a) For the dark-colored pixel, its eight neighbors are examined to see if any silhouette curve passes it. (b) With an image-space method, such a thin gap between two objects with similarly shaded colors is hard to detect, resulting in ugly spatial/temporal aliases.*

An important key to effective antialiasing is how to efficiently and robustly find those troublesome pixels that such silhouette curves may cross. In previous adaptive supersampling methods, sampled pixel colors were often adaptively compared between pixels to decide where to take extra samples. When, for instance, three color disparities were found for the dark-colored pixel in the north, northwest, and west directions, as in Figure 2(a), more samples were usually taken systematically around the pixel's upper left corner. One problem with such an adaptive computation is that it entails an unpredictable, data-dependent control structure that could degrade the computation performance on today's many-core processors in which warps of threads are processed in SIMD fashion.

For a simpler decision algorithm that is better suited for GPU computing, we subdivided each image pixel into four subpixels and independently performed a test with simple control flow against each subpixel, deciding if its subpixel region needs extra sampling. Figure 3 depicts two possible extreme situations that occur when a decision test is applied. The figure in (a) represents an *optimistic* situation in which all four subpixels of the pixel are covered entirely by one object, requiring no further effort for supersampling in the point of silhouette curve. On the other hand, the figure in (b) illustrates a *pessimistic* situation where three of them, except for the one in the fourth quadrant, are crossed by a boundary curve.



(a) An *optimistic* case (b) A *pessimistic* case

Figure 3: *Two extreme situations. Usual cases fall between these extremes.*

To achieve a simple control flow in this examination process, we take the pessimistic approach that always assumes the pessimistic case. Then, the decision for a given subpixel can be made simply by comparing color attributes of three neighboring pixels respectively with the pixel it belongs to, where the locations of the neighbors are easily determined depending on the subpixel's location, as shown in Figure 4. When there is at least one disparity under a given threshold criterion, the subpixel is regarded as one that needs special treatment.

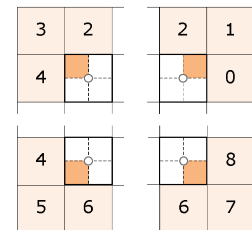


Figure 4: *Three neighbors for attribute comparison.*

4.1.2 Presampling for building three two-dimensional arrays

While it usually works fine, the test based on color measure alone often fails to detect subtle aliases that occur in a sophisticated manner (for instance, see Figure 2(b)). Another problem is that it is hard to effectively handle a variety of sources of rendering artifacts resulting from ray tracing with a single threshold τ_{col}. In order to complement such an image-space method, we additionally explore the per-pixel information on objects that each primary ray hits. Combined with the color measure, such an object space geometry attribute can generate a synergistic effect in robustly and selectively reducing rendering aliases.

For efficient computation in the later stages, we first build three two-dimensional arrays with the same resolution by presampling image pixels at their centers through ray tracing. The first one to be constructed is *the shaded color image* that stores ray-traced, shaded

colors. The second one is *the color reference map* that contains pixel values that are referred to when a test for color disparity between pixels is performed. While several measures such as luminance or the length of the gradient vector obtained by the Sobel operator are possible as color references, our current implementation simply stores the shaded colors in this map, and applies a contrast criterion in the following stage, as will be explained shortly. The last image produced is *the geometry attribute map*, in which each pixel stores the identification number of an object (Object ID) that was hit by the primary ray fired thorough its center. The first array is used as an accumulation buffer into which extra sampled colors are accumulated in the final stage of computation. On the other hand, stored in texture form, the other two maps provide both color reference and geometry attribute information through which an effectively controlled decision can be made for selective antialiasing.

4.1.3 Two-step subpixel test

Once the three arrays are ready, a two-step test is performed for each subpixel to see if more sampling rays are necessary for its region. The purpose of the first step is to choose a threshold that will be used in the next color reference comparison step. We first perform a comparison operation with three neighbors using the geometry attribute, that is, the Object ID in this simple example. As the pessimistic approach is taken in our method, a current subpixel is regarded as *possibly problematic with respect to the geometry attribute* if there is at least one mismatch between their Object IDs. When that happens for a given subpixel, there is a chance that a silhouette curve may cross its region, possibly causing annoying artifacts. In order to treat such possibly problematic subpixels separately, we assign a different threshold $\tau_{oid} \in [0, 1]$ that is selectively controlled by the user. On the other hand, we set a threshold $\tau_{col} \in [0, 1]$ to all other subpixels for which no possible problems are found with respect to the object space information. Then, another neighbor comparison operation is carried out using the color reference based on the assigned threshold. Whichever threshold is applied, the current subpixel is considered as *problematic* and is marked as *active* if there is at least one color disparity with respect to the threshold.

Note that the key idea of our method is to place a separate threshold τ_{oid} on the regions of subpixels that are possibly problematic with respect to the geometry attribute instead of applying an identical threshold τ_{col} to entire subpixel regions. By using a stricter value of τ_{oid}, it is possible to selectively focus computational effort more on reducing aliases due to the geometry attribute (when it is set to zero, those pixels are always supersampled). Because the pessimistic sampling strategy is taken, some subpixels that actually need no extra sampling may be supersampled. However, the presented simple, parallely executable adaptive scheme allows an efficient implementation on the current GPU architecture, as will be explained.

4.1.4 Subpixel sampling and color summing

For each subpixel found to be problematic, four extra samples are taken by shooting respective rays in a stratified jittered fashion. The leftmost figure in Figure 5 illustrates an example where three of the four subpixels are found to be active. When extra sampling is done for all problematic subpixels, their subsampled colors are then summed with those of the inactive subpixels. In our scheme, the shaded pixel color that has been computed at its center in the presampling computation is transferred to its inactive subpixels, each weighted by $\frac{1}{4}$. This can be implemented by multiplying a proper weight, decided by the number of inactive subpixels, to the corresponding pixel color in the shaded color image. Then, the color summing operation is carried out simply by accumulating each supersampled color with a weight $\frac{1}{16}$ to its pixel location. Once all accumulation is over, the shaded color image turns into one containing an antialiased rendering image.

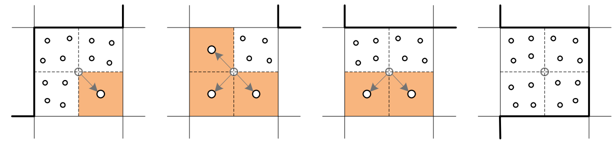


Figure 5: *Four examples of adaptive sampling. The color of a pixel, sampled at its center in the presampling stage, is used for its inactive subpixels, if any.*

4.2 Extension of the basic idea

4.2.1 Color reference and geometry attributes

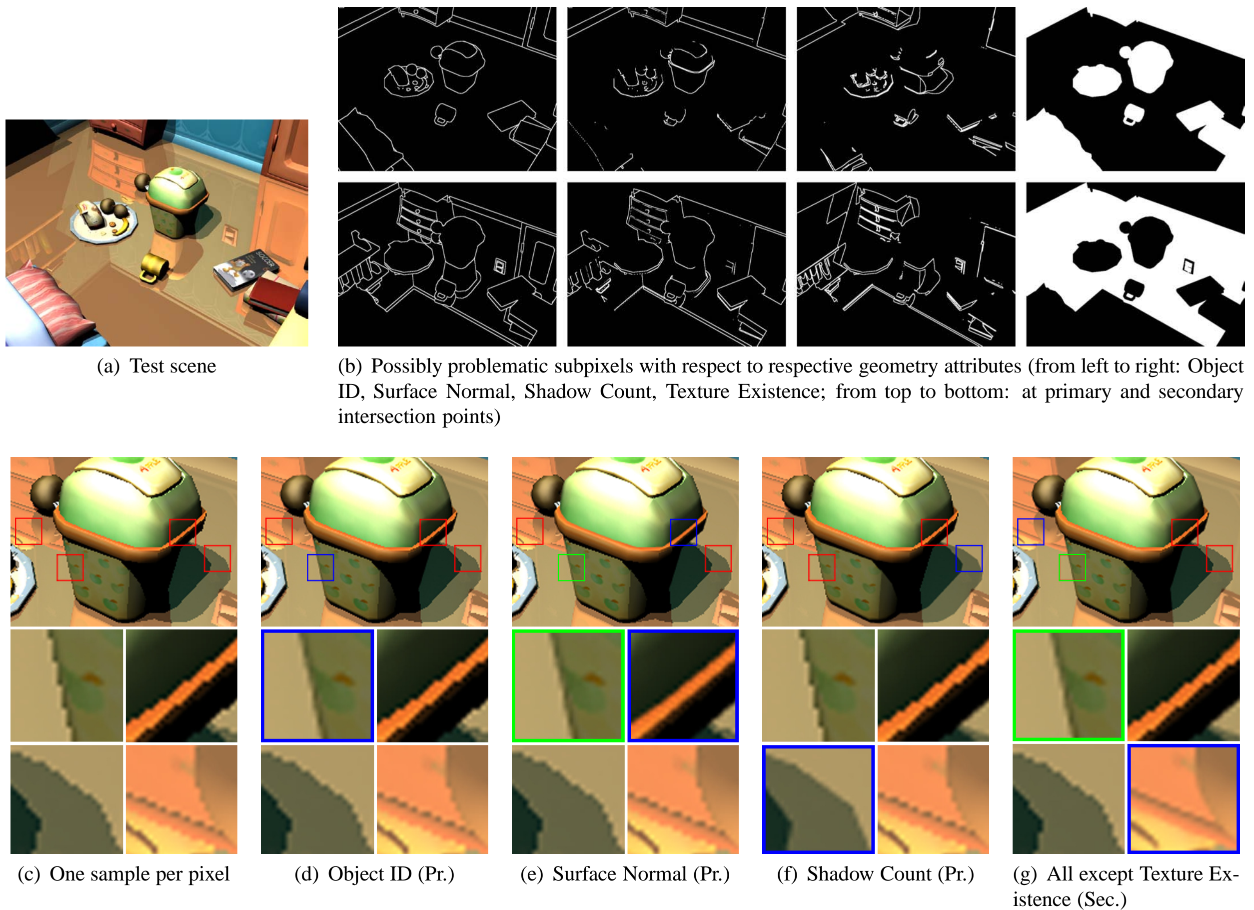
The presented idea is naturally extended to include additional geometry attributes that are relevant to ray tracing. Table 1 lists per-pixel attributes that are explored in our method. Basically, the *Color Reference*, which is used to investigate color differences between adjacent pixels, is the main attribute that eventually determines if a given subpixel needs extra sampling. In our current implementation, the shaded pixel color is used as a color reference.

Collection point	Attribute	Threshold
At pixel center	*Color Reference*	τ_{col}
At primary ray hit point	*Object ID*	τ_{poid}
	Surface Normal	τ_{psn}
	Shadow Count	τ_{psc}
	Texture Existence	τ_{pte}
At secondary ray hit point	*Object ID*	τ_{soid}
	Surface Normal	τ_{ssn}
	Shadow Count	τ_{ssc}
	Texture Existence	τ_{ste}

Table 1: *Three classes of pixel attributes explored in our method. In our current implementation, four object-space measures were considered for robust antialiasing to compensate for shaded color, a typical image-space attribute used in previous works. Users can selectively choose attributes and control their threshold values in order to emphasize specific features in antialiasing.*

The other items are geometry attributes, which are deeply related to various types of rendering artifacts produced by recursive ray tracing. First, four types of geometry attributes are collected at the intersection point hit by the primary ray. The *Object ID*, explained in the previous subsection, is often the most serious source of annoying aliases. The *Surface Normal* remembers the normal direction of a surface at the primary ray hit point, and is particularly useful for detecting such an edge formed by polygons of an object that meet at an acute angle. The *Shadow Count* records the number of light sources that are invisible from the intersection point, and is used to reduce artifacts that occur along shadow boundaries. Finally, the *Texture Existence* is a Boolean variable that indicates whether any texture is applied at the intersection point. By keeping this information, it is possible to apply a different criterion for controlling texture antialiasing.

Another set of the four geometry attributes are gathered at the respective surface points hit by the secondary, reflection/refraction rays that produce such nice rendering effects as reflection and refraction. The rendering quality for these secondary effects can be



(a) Test scene

(b) Possibly problematic subpixels with respect to respective geometry attributes (from left to right: Object ID, Surface Normal, Shadow Count, Texture Existence; from top to bottom: at primary and secondary intersection points)

(c) One sample per pixel (d) Object ID (Pr.) (e) Surface Normal (Pr.) (f) Shadow Count (Pr.) (g) All except Texture Existence (Sec.)

Figure 6: *Examples of selective supersampling. In the test scene shown in (a), textures are applied to all diffusive objects except the floor, which is the only specular object; (b) shows detected subpixels that are possibly problematic. In (c), annoying rendering artifacts are easily seen in magnified portions of a ray-traced image obtained by taking one sample per pixel. Images in (d) to (g) that were generated by respectively setting threshold values of selected geometry attribute(s) to very small values demonstrate that the idea of selective supersampling works well. In these comparisons, the blue box indicates regions detected by selected attribute(s) only, whereas the green box indicates regions that were also found by other attributes.*

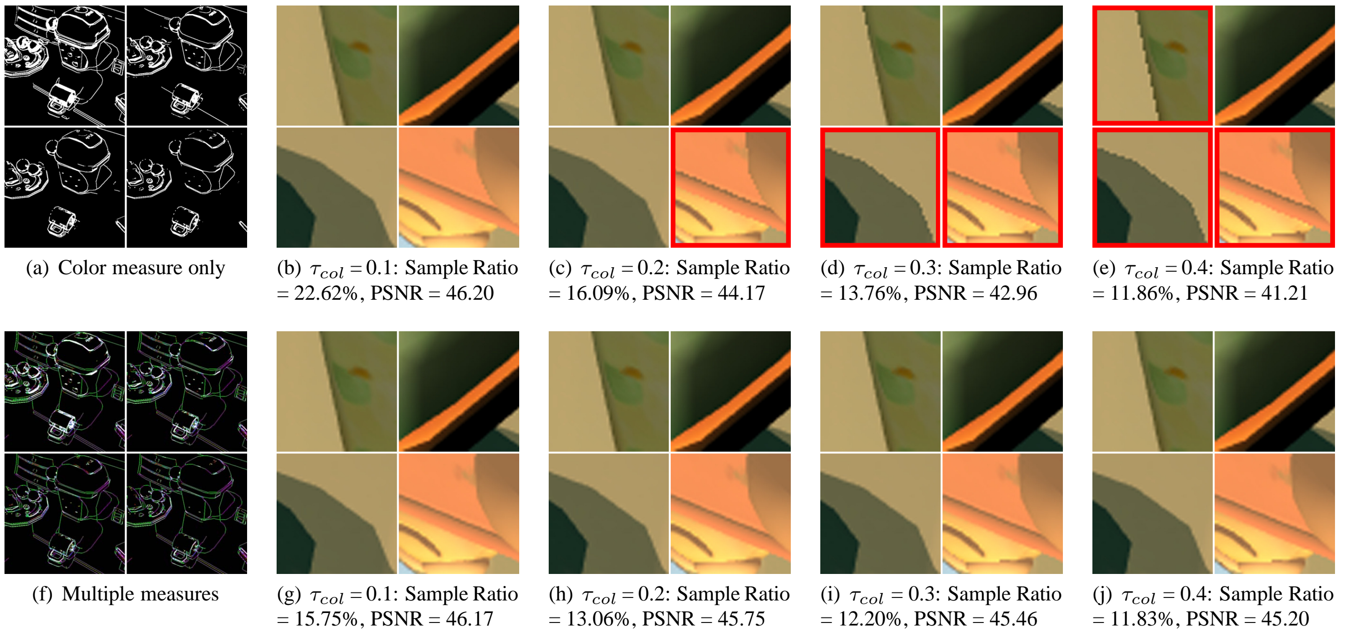
enhanced effectively by selectively handling the secondary geometry attributes rather than attempting to reduce aliases after they are blended into final shaded colors. In order to take care of all secondary rays spawned, a geometry attribute quadruple would have to be produced per spawned ray. In our implementation, however, only one-bounced secondary rays are considered, as they are often sufficient in real-time ray tracing, whereas the given idea permits a natural extension to multiple bounces. Once all necessary attributes are gathered for each pixel, they are stored in the geometry attribute map. Notice that depth information, which is yet another type of possible geometry information, has often been used for an antialiasing purpose. While it is also effective, we find that the presented geometry attributes effectively replace it.

4.2.2 Detection of problematic subpixels

Unlike when using Object ID only, the geometry attribute comparison becomes a little bit complicated, as each pixel is now associated with a vector of geometry attributes. When the first step comparison operation is performed for a given subpixel, item-to-item comparisons with three neighbors are carried out to collect a list of disagreeing geometry attributes. The integer-valued Object ID and Shadow Count attributes are said to differ from each other when pixels have different values. The Surface Normal attributes of two pixels are regarded as different if their cross product is less than a preset value. Finally, the Texture Existence field is always set on regardless of neighbors when the current subpixel's attribute is true.

When discordance is found for at least one geometry attribute, the subpixel is considered as possibly problematic and the smallest value of the thresholds set to the disagreeing attributes is used in the next color reference comparison. Otherwise, the threshold τ_{col} set to the Color Reference attribute is used instead. In the next color reference comparison that actually decides if the subpixel is problematic, we apply the contrast measure proposed by Mitchell [Mitchell 1987]. In the original method, (0.4, 0.3, 0.6) was used as the default threshold vector whose respective values are applied to the corresponding contrast values $I_\lambda = \frac{I_\lambda^{max} - I_\lambda^{min}}{I_\lambda^{max} + I_\lambda^{min}}$ ($\lambda = R, G, B$). In order to provide a user with controllability in adaptive sampling, we apply $(\tau \cdot 1.36, \tau \cdot 1.02, \tau \cdot 2.04)$ as a threshold vector, where $\tau \in [0, 1]$ is the threshold that was set according to the result of geometry attribute comparison (note that it becomes roughly $(0.4, 0.3, 0.6)$ when $\tau = 0.3$).

Before rendering, the user can selectively set the nine threshold val-



(a) Color measure only

(b) $\tau_{col} = 0.1$: Sample Ratio = 22.62%, PSNR = 46.20

(c) $\tau_{col} = 0.2$: Sample Ratio = 16.09%, PSNR = 44.17

(d) $\tau_{col} = 0.3$: Sample Ratio = 13.76%, PSNR = 42.96

(e) $\tau_{col} = 0.4$: Sample Ratio = 11.86%, PSNR = 41.21

(f) Multiple measures

(g) $\tau_{col} = 0.1$: Sample Ratio = 15.75%, PSNR = 46.17

(h) $\tau_{col} = 0.2$: Sample Ratio = 13.06%, PSNR = 45.75

(i) $\tau_{col} = 0.3$: Sample Ratio = 12.20%, PSNR = 45.46

(j) $\tau_{col} = 0.4$: Sample Ratio = 11.83%, PSNR = 45.20

Figure 7: *Single- versus multiple-valued thresholds for adaptive supersampling. Images in the first row were produced by varying τ_{col}, the threshold for Color Reference, with all geometry attributes turned off. As τ_{col} increased, the rendering went faster as fewer extra ray samples were taken. However, with this color measure alone, rendering artifacts became more visible regardless of image content. On the other hand, the second row images, created with geometry attributes turned on, demonstrate that by properly setting thresholds for geometry attributes, we were able to distribute more ray samples adaptively and selectively to subpixel regions that caused visually annoying artifacts. As clearly observed, the decrease in the PSNR (Peak Signal-to-Noise Ratio) value was minimized, although fewer samples were taken. In the captions, 'Sample Ratio' indicates the ratio of the number of ray samples taken to that used when 16 samples were taken per pixel. The images in (a) and (b) respectively display color-coded images of problematic subpixels in which the colors black, red, green, blue, and white, respectively, indicate the number of subpixels in the pixel that were found to be problematic (0 to 4 in increasing order).*

ues of Table 1 in the interval of $[0, 1]$, adjusting the importance of respective rendering features applied to the antialiasing computation. For example, if shadows were a critical element in the current rendering, a higher fraction of computing time could be directed to shadow antialiasing by setting rigorous threshold values for the two shadow attributes and turning off some other low-priority attributes by setting their thresholds to one. It should be noted that it is not easy to realize this selective antialiasing capability by image-space, color measure alone. (See Figure 6 and 7 for examples).

4.3 Implementation on the GPU architecture

The presented supersampling method is well suited to implementation on the CUDA computing architecture. Conceptually, numerous pixels or subpixels processed during rendering correspond to computational threads to each one of whom a sequence of kernels are run independently with minimized communication. In our GPU implementation, the *presampling* and *subpixel test* stages were carried out by two kernels: `Presampler` and `ActSubPixDetector`. Then, the following *subpixel sampling* and *color summing* stages were performed by the `ExtraSampler` kernel.

4.3.1 `Presampler` kernel

Suppose that we are to render an image of $m \times n$ pixels. Before starting the presampling computation, memory spaces for three arrays `ShadedColorImage`, `ColoreReferenceMap`, and `GeometryAttributeMap` are allocated in global memory. In our implementation, 32 bytes are used per pixel to store geometry attributes (2, 12, 1, and 1 byte(s) for each of Object ID, Surface Normal, Shadow Count, and Texture Existence, respectively). The rendering session starts by running the `Presampler` kernel with thread blocks of size $m_p \times n_p$, where the blocks correspond to image tiles of $m_p \times n_p$ pixels (our test shows that the presampling computation runs fastest on the 4×64 block size). For each thread, it performs the regular ray tracing against the assigned pixel, storing geometry attributes into the pixel's location at `GeometryAttributeMap`. When the tracing is over, the kernel also writes the same final shaded color twice, respectively into `ShadedColorImage` and `ColoreReferenceMap`.

Unlike the `ShadeColorImage` array that functions as an accumulation buffer in global memory, `ColorReferenceMap` and `GeometryAttributeMap` are treated as texture images in texture memory that provides efficient cached access to the next kernel.

4.3.2 `ActSubPixDetector` kernel

The next running `ActSubPixDetector` kernel explores the collected color reference and geometry attributes to locate problematic subpixels, storing them in an array in global memory, called `ActSubPixBuffer`. In this stage, the image plane is partitioned into a set of image tiles with $m_a \times n_a$ pixels, and thread blocks of $2m_a \times 2n_a$ threads are launched to run this kernel, where the threads of a block represent subpixels of the corresponding tile. For efficient computation, a $2m_a \times 2n_a$ array, called the `ActiveOnes` is allocated per block in shared memory to temporarily record which one in the block is marked as active (see Figure 8).

From the thread ID, the kernel knows which subpixel (and the pixel that it belongs to) it is currently processing, and computes indices

of to-be-compared neighboring pixels by accessing a simple offset table that holds respective offsets (this table is small, hence it can be located in the constant or texture memory that provides fast cached access). Then, it compares the current subpixel's geometry attributes with those of the three relevant neighbors by accessing the `GeometryAttributeMap` texture.

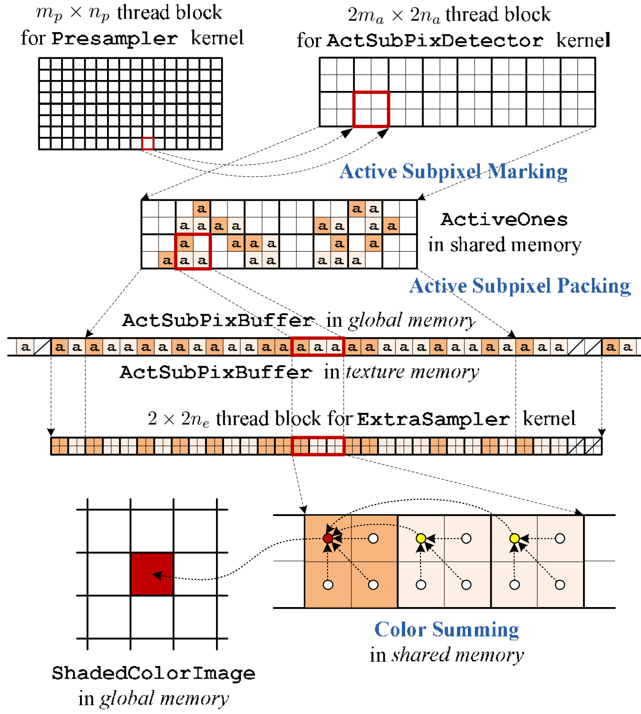


Figure 8: *Data structures for our GPU implementation.*

The kernel additionally fetches four color references (shaded colors) from the same locations of the `ColorReferenceMap`, and checks their contrasts using the threshold that was set properly according to the geometry attribute comparison. When the current subpixel is found as problematic, the kernel marks the subpixel's location in `ActiveOnes`. When the marking operation is completed for all threads in the same block, two additional computations follow.

One is to prepare the `ShadedColorImage` array for the extra sampling computation performed by the `ExtraSampler` kernel. As mentioned before, the supersampled colors will simply be added to the accumulation buffer at the later stage; hence, the weight of inactive subpixels should have been reflected appropriately before then. To solve this problem, the first thread of each four-thread group, corresponding to a pixel, is elected as representative, and counts the number of active subpixels, n_{asp} in its parent pixel and multiplies $\frac{4-n_{asp}}{4}$ into the pixel's shaded color in `ShadedColorImage` (recall Figure 5).

The second task to be performed is to build the `ActSubPixBuffer` in global memory, so that the next `ExtraSampler` kernel knows through the texture fetch where to shoot extra sample rays. One possible solution would be to copy the contents of the `ActiveOnes` array in shared memory to `ActSubPixBuffer` and have the `ExtraSampler` kernel process blocks of the buffer. Such a simple approach, however, leads to computational inefficiency, because the buffer is usually marked sparsely, frequently producing warps during the CUDA processing that contain many *null* threads that need no extra sampling (see Figure 8 for an example of sparsely marked subpixels in the `ActiveOnes` array).

In an aim to have processing cores of the GPU work in parallel as much as possible, we pack active subpixels when they are stored into the `ActSubPixBuffer` array. To implement the packing operation, the first thread of the entire $2m_a \times 2n_a$ block is now chosen as representative, and counts the total number of active subpixels in the block while packing their indices. Then, the kernel appends the packed index information to the end of `ActSubPixBuffer`. The index-copying operation from the shared to global memory must be handled carefully, because several representative threads from different parallel blocks may attempt to access the same buffer space, possibly causing data loss. Fortunately, the CUDA API (Compute Capability 1.1) provides a read-modify-write *atomic add* operation that reads a 32-bit integer variable in global memory, adds an integer to it, and writes the result back to the same variable [NVIDIA 2008]. By keeping a pointer variable in global memory that points to the next available memory space in `ActSubPixBuffer`, the concurrent representatives can allocate necessary memory space harmoniously without interference between blocks.

It should be noted that it may seem inefficient for one representative thread to scan all $2m_a \times 2n_a$ elements of the `ActiveOnes` array while the remaining threads in the block are waiting. It is possible to have each thread corresponding to an active subpixel write its own subpixel index to `ActSubPixBuffer` via the atomic add operation. However, such a method would create a large number of atomic add operations, which would deteriorate performance significantly. Furthermore, as many concurrent threads that contend for the same atomic pointer variable access it in unpredictable order, the active subpixels are stored in the buffer somewhat randomly, losing their coherency. Because keeping spatial locality between sample rays within a warp is important, the low coherence in the `ActSubPixBuffer` could cause a severe efficiency problem in the next extra sampling stage. For a block size of 8×16 ($2m_a \times 2n_a$) used in our implementation where 128 elements are scanned by the representative thread, the timing comparison shows that the presented packing method runs markedly faster than the other possible implementations. One last thing to mention is that when the indices of active subpixels are stored, the first active subpixel in each pixel is specially marked as a master (the darker colored subpixels in Figure 8), so that threads of masters respectively collect extra sampled color in the next stage.

4.3.3 `ExtraSampler` kernel

In the previous computation, active subpixels have been serialized into `ActSubPixBuffer`, so there exists only linear locality between subpixels. As such, we view this buffer as a one-dimensional array, and partition it into $1 \times n_e$ chunks ($n_e = 64$ in our current implementation). The final subpixel sampling stage proceeds by running the `ExtraSampler` kernel with thread blocks of size $2 \times 2n_e$, where one quadruple of threads handles extra sampling over one active subpixel. As before, for an efficient color summing operation, an array of $2 \times 2n_e$ colors are allocated in shared memory to temporarily hold subsampled colors. Given the thread ID, the kernel decides the ray direction using subpixel index information from `ActSubPixBuffer` and a modular-four operation, performs ray tracing, and stores shaded colors in the shared memory location. When all ray tracing operations are done, the color summing operation proceeds hierarchically as follows (see Figure 8 again): the first thread of each thread quadruple for active subpixels sums all four subsampled colors and stores the result at its location. Then, the first thread of the quadruple that has been marked as a master subpixel, designated in the active subpixel

marking computation, collects all the needed colors and accumulates the summed color, multiplied by $\frac{1}{16}$, into the pixel location of `ShadedColorImage`.

While this color summing on shared memory is quite efficient, one thing must be taken into account. In the CUDA computing structure, only threads in the same block can share data in shared memory. This means that, for a successful color sum operation, all subpixels from the same pixel must exist in the same thread block. As such, our implementation has aligned active subpixels properly along the $1 \times n_e$ boundaries when they were put into the `ActSubPixBuffer` array.

5 Experimental results

To show its effectiveness, we implemented our method and tested it using several examples on the NVIDIA's GeForce GTX 280 processor. In developing our GPU ray tracer, we employed the short stack method for kd-tree traversal [Horn et al. 2007]. The kernels of our ray tracer consumed up to 58 registers. For the tested GPU that has 16,384 registers and 16 Kbytes of shared memory per multiprocessor, we could allocate up to 7 stack elements of 8 bytes each per thread in shared memory after a small amount of space was saved for bookkeeping (note that the required memory spaces for the short stack and the temporary storage described in the previous section have nonoverlapping live ranges).

Table 2 compares our sampling method to fixed density (i.e., non-adaptive) supersampling where densities of 1, 4, 9, and 16 samples per pixel are considered. To build the ray tracer performing the fixed-density supersampling, we modified the adaptive sampling part of our ray tracer. For all scenes except Fairy Forest, we adjusted the thresholds for color reference and geometry attributes so as to achieve PSNR values that fell between those of densities of 9 and 16 samples per pixel. At 1 or 4 samples per pixel, annoying aliasing artifacts were clearly visible. On the other hand, at 9 or 16 samples per pixel, aliases were barely visible as was the same for our adaptive sampling.

As the timing results indicate, our sampling method is quite favorable in that it is usually 2 to 3 times faster than the nonadaptive sampling of 9 samples per pixel while producing renderings of slightly better image quality except for the Fairy Forest example. This efficiency is due in part to high efficiency in the decision process that determines where and how many ray samples are made for antialiasing, where only around 20% of samples were taken for the first four scenes compared to the density of 9 samples per pixel (the subpixel test stage usually took less than 25 ms for all tested scenes). For the Fairy Forest scene that contains highly detailed textures, stricter thresholds had to be used to achieve a high PSNR value (1.28 fps). Using our selective scheme, however, we could adjust them so that more ray samples were directed to perceptually more visible artifacts where the thresholds $\tau_\alpha = 0.6$ offered an image quality similar to (or slightly better than in some selected regions) the Fixed 9 sampling (1.86 fps) while achieving about two times of speedup (3.75 fps). Although the PSNR value was lower, the noise from textures was barely visible in our result.

Our method easily allows further extensions to various situations. For instance, suppose that a camera rotates around the fairy while focusing on her. In this case, the viewer's attention is usually focused on the fairy, and it is quite desirable to allocate more of the very limited computing to sampling the area to which the objects for the fairy, grass, and dragonfly are projected and less to the remaining region. To prove its effectiveness, we slightly modified the attribute comparison operation in such a way that the geometry attributes between pixels differ only when the designated Object IDs are involved. As a result, we could gain about 2.7 times of speedup (5.02 fps) while maintaining a high quality for the selected objects. See Figure 1(g) to (j).

6 Concluding remarks

We have presented a selective and adaptive supersampling technique that was designed for real-time ray tracing on many-core processors, and demonstrated its effectiveness through several examples. The implementation on the GPU has shown that the proposed supersampling technique can be run effectively on the current, massively parallel streaming SIMD processor. In the presented method, nearest filtering was applied for each inactive subpixel as the color of its parent pixel, taken at the center, is used (see Figure 5 again). As an option for texture filtering, bilinear filtering can be included by slightly modifying the current implementation, in which the inactive subpixel's color is computed from its four neighboring pixels' colors, easily accessible from the color reference map. Our test shows that this extension requires only little extra cost, and reduces texture aliasing slightly. The issue of combining an appropriate texture filtering technique with our sampling method is left as a future research topic.

In addition to the GPU, we are currently implementing our method on the current multicore CPUs, where a preliminary experimental result implies that it is as effective as on the GPU. We believe that the supersampling algorithm presented here will also be easily mapped on the upcoming many-core CPUs, such as the Intel Larrabee processor, offering an effective supersampling scheme for the developers of real-time ray tracers.

References

AMANATIDES, J. 1984. Ray tracing with cones. *Proceedings of SIGGRAPH 1984 18*, 3, 129–135.

BOLIN, M., AND MEYER, G. 1998. A perceptually based adaptive sampling algorithm. In *Proceedings of SIGGRAPH 1998*, 299–309.

COOK, R., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. In *Proceedings of SIGGRAPH 1984*, 131–145.

CROW, F. 1977. The aliasing problem in computer-generated shaded images. *Communications of the ACM 20*, 11, 799–805.

FOLEY, T., AND SUGERMAN, J. 2005. KD-tree acceleration structures for a GPU raytracer. In *Proceedings of HWWS 2005*, 15–22.

GENETTI, J., GORDON, D., AND WILLIAMS, G. 1998. Adaptive supersampling in object space using pyramidal rays. *Computer Graphcis Forum 17*, 1, 29–54.

HECKBERT, P., AND HANRAHAN, P. 1984. Beam tracing polygonal objects. In *Proceedings of SIGGRAPH 1984*, 119–127.

HORN, D., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive k-D tree GPU raytracing. In *Proceedings of I3D 2007*, 167–174.

LONGHURST, P., DEBATTISTA, K., GILLIBRAND, R., AND CHALMERS, A. 2005. Analytic antialiasing for selective high fidelity rendering. In *Proceedings of SIBGRAPI 2005*, 359–366.

MITCHELL, D. 1987. Generating antialiased images at low sampling densities. In *Proceedings of SIGGRAPH 1987*, 65–72.

NVIDIA. 2008. *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide (Version 2.0).*

OHTA, M., AND MAEKAWA, M. 1990. Ray-bound tracing for perfect and efficient anti-aliasing. *The Visual Computer 6*, 3, 125–133.

OVERBECK, R., RAMAMOORTHI, R., AND MARK, W. 2008. Large ray packets for real-time Whitted ray tracing. In *IEEE/EG Symposium on Interactive Ray Tracing*, 41–48.

POPOV, S., GÜNTHER, J., SEIDEL, H.-P., AND SLUSALLEK, P. 2007. Stackless KD-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum (Proceedings of Eurographics) 26*, 3, 415–424.

SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. 2008. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics 27*, 3, 1–15.

SHEVTSOV, M., SOUPIKOV, A., AND KAPUSTIN, A. 2007. Highly parallel fast KD-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum (Proceedings of Eurographics) 26*, 3, 395–404.

THOMAS, D., NETRAVALI, A., AND FOX, D. 1989. Antialiased ray tracing with covers. *Computer Graphcis Forum 8*, 4, 325–336.

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1, 6.

WALD, I., MARK, W., GÜNTHER, J., BOULOS, S., IZE, T., HUNT, W., PARKER, S., AND SHIRLEY, P. 2007. State of the Art in Ray Tracing Animated Scenes. In *Eurographics 2007 State of the Art Reports*.

WALD, I., BENTHIN, C., AND BOULOS, S. 2008. Getting rid of packets – efficient SIMD single-ray traversal using multi-branching BVHs. In *IEEE/EG Symposium on Interactive Ray Tracing*, 49–57.

WALD, I. 2004. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University.

WHITTED, T. 1980. An improved illumination model for shaded display. *Communications of the ACM 23*, 6, 343–349.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time KD-tree construction on graphics hardware. *ACM Transactions on Graphics 27*, 5, 1–11.

Scene	Sampling	PSNR in dB	Framerate in fps
Bathroom (268K)	Fixed 16	46.35	0.79 (8.68)
	Ours ($\tau_\alpha = 0.6$)†	**44.03 (24.6%)**	**2.14 (3.22)**
	Fixed 9	**43.47**	**1.23 (5.62)**
	Fixed 4	39.36	2.36 (2.92)
	Fixed 1	35.06	6.89 (1.00)
Conference (190K)	Fixed 16	47.43	1.39 (9.54)
	Ours ($\tau_\alpha = 0.6$)	**46.67 (19.8%)**	**4.58 (2.88)**
	Fixed 9	**44.48**	**2.28 (5.81)**
	Fixed 4	40.54	4.52 (2.92)
	Fixed 1	36.43	13.23 (1.00)
Room (117K)	Fixed 16	47.61	1.19 (9.96)
	Ours ($\tau_\alpha = 0.6$)	**44.56 (17.4%)**	**5.72 (2.07)**
	Fixed 9	**44.31**	**1.92 (6.15)**
	Fixed 4	39.76	3.92 (3.02)
	Fixed 1	35.98	11.83 (1.00)
Kitchen (101K)	Fixed 16	46.78	1.24 (10.12)
	Ours ($\tau_\alpha = 0.6$)	**44.16 (22.4%)**	**4.46 (2.81)**
	Fixed 9	**43.85**	**2.01 (6.22)**
	Fixed 4	39.72	4.09 (3.07)
	Fixed 1	35.58	12.53 (1.00)
Fairy Forest (174K)	Fixed 16	47.22	1.16 (9.20)
	Ours ($\tau_\alpha = 0.1$)	**44.11 (106.5%)**	**1.28 (8.34)**
	Fixed 9	**43.88**	**1.86 (5.74)**
	Ours ($\tau_\alpha = 0.3$)	39.92 (28.6%)	2.82 (3.78)
	Fixed 4	39.19	3.68 (2.90)
	Ours ($\tau_\alpha = 0.6$)	**38.61 (19.5%)**	**3.75 (2.84)**
	Ours ($\tau_\alpha = 1.0$)‡	37.80 (16.5%)	4.51 (2.36)
	Ours⋆	**36.05 (13.5%)**	**5.02 (2.12)**
	Fixed 1	35.00	10.68 (1.00)

† $\tau_\alpha = \tau_{col}, \tau_{pte}, \tau_{ste}$, ‡ Plus $\tau_{psc} = \tau_{ssc} = 1.0$, ⋆ Fairy focused.

Table 2: *Performance comparison with fixed-density supersampling. All scenes were rendered with shading, textures, reflection/refraction, and shadows at resolution* 1024×1024*. The PSNR values were measured by comparing the respective results with ground-truth images produced by taking 256 nonadaptive samples per pixel. The figures in parentheses in the PSNR column mean the ratios of the number of ray samples actually taken by our method to that of the Fixed 9 sampling. On the other hand, the figures in parentheses in the Frame rate column indicate the overheads relative to the case of one sample per pixel. For a fair comparison, we applied a similar shared memory technique for an efficient implementation of fixed-density supersampling, where the overheads were measured in the range of 8.68 to 10.12 for the Fixed 16 sampling. The default thresholds applied are* $\tau_{poid} = \tau_{soid} = 0.05$ (0.0 *for Conference),* $\tau_{psn} = \tau_{ssn} = 0.06$, $\tau_{psc} = \tau_{ssc} = 0.06$.