Eurographics Symposium on Rendering (2022)
A. Ghosh and L.-Y. Wei (Guest Editors)

Multi-Fragment Rendering for Glossy Bounces on the GPU

Atsushi Yoshimura Yusuke Tokuyoshi Takahiro Harada

Advanced Micro Devices, Inc.

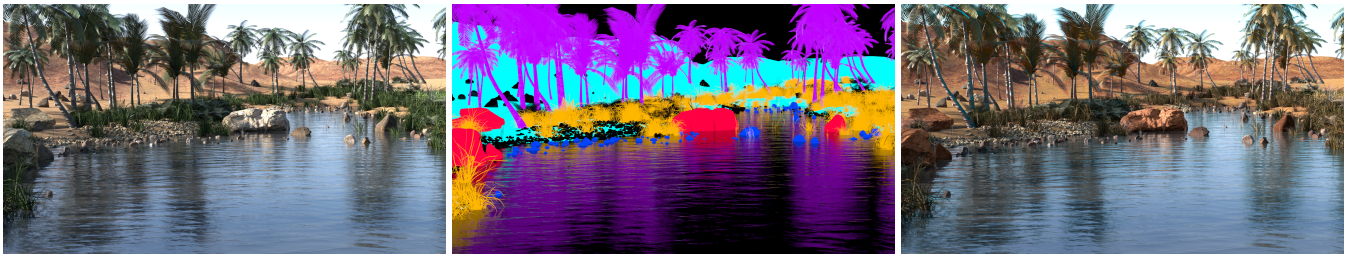


Figure 1: *Left: Rendered image. Middle: Visualization of object IDs (color) and their coverages (brightness) that are generated using our method for glossy reflections. Note that indirect visibility is captured on the water surface. Right: Re-coloring using the IDs and coverages after rendering.*

Abstract

Multi-fragment rendering provides additional degrees of freedom in postprocessing. It allows us to edit images rendered with antialiasing, motion blur, depth of field, and transparency. To store multiple fragments, relationships between pixels and scene elements are often encoded into an existing image format. Most multi-fragment rendering systems, however, take into account only directly visible fragments on primary rays. The pixel coverage of indirectly visible fragments on reflected or refracted rays has not been well discussed. In this paper, we extend the generation of multiple fragments to support the indirect visibility in multiple bounces, which is often required by artists for image manipulation in productions. Our method is compatible with an existing multi-fragment image format such as Cryptomatte, and does not need any additional ray traversals during path tracing.

CCS Concepts

• ***Computing methodologies*** → *Image manipulation; Ray tracing;*

1. Introduction

Multi-fragment rendering [VVP20] has been studied for a large variety of applications such as order-independent transparency and antialiasing. It is also applicable to matte generation with an arbitrary combination of objects. This is achieved by storing the pair of an object identifier (e.g., object ID and material ID) and it's coverage for each pixel into an image format such as Cryptomatte [Psy15]. To handle multiple fragments in a single pixel, most of the existing works focused only on directly visible fragments on primary rays for antialiasing, transparency, and perfectly specular reflections or refractions. However, a matte image for objects reflected or refracted on glossy surfaces is often required for composition and re-coloring in productions. Without such information of indirectly visible fragments, we cannot capture objects through multiple glossy bounces in a post process (see Fig. 5b).

In this paper, we present an efficient method to generate fragments scattered from glossy (or perfectly specular) surfaces that fragments can be represented as multi-fragment image formats such as Cryptomatte. To compute the coverage for each identifier in multiple glossy bounces, we introduce a weighting function for the coverage to attenuate the visibility according to the diffusion of rays. We also present our GPU implementation that stores fragments in a fixed-size storage considering the priority of fragments. Using our method, we are able to obtain a matte image for glossy reflections and refractions as shown in Fig. 1. To summarize, the contributions of this paper are as follows.

- We introduce a computation method for fragment coverage in multiple bounces using a weighting function (§ 3.1).
- We present a practical weighting function that penalizes the coverage of a fragment with less visibility of object details (§ 3.2).
- This paper also describes the implementation details of our coverage update process on the GPU (§ 4).

2. Related Work

The seminal work regarding multiple depth fragments was A-buffer [Car84]. Saito and Takahashi [ST90] discussed an extension of the A-buffer for reflections and refractions to render contour lines in postprocessing. For a comprehensive survey of multi-fragment rendering techniques and applications, we refer the reader to Vasilakis et al. [VVP20]. To store generated multiple fragments into a file, Cryptomatte represents a fragment with the pair of an object identifier (ID) and coverage for each pixel, and then stores these ID–coverage pairs into an arbitrary number of color components [FJ15]. In this paper, we use this Cryptomatte format to store our multiple fragments generated for glossy bounces.

3. Multi-fragment Rendering for Glossy Bounces

Since visual details of reflected or refracted objects can be lost by the diffusion of rays, fragments on such scattered rays have often been ignored in multi-fragment rendering. Instead of fully ignoring such fragments, we store them by reducing the coverage according to the amount of diffusion.

3.1. Our Coverage Computation

The coverage of a fragment is the percentage of rays not passing through the surface. In this paper, we treat a glossy bounce as a passing through while reducing the amount of the passage according to the diffusion of rays. We propose to obtain this coverage by a recursive form similar to the rendering equation as follows:

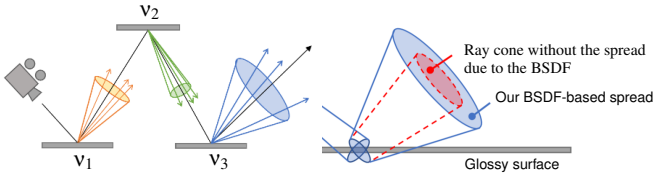
$$C_i(\mathbf{x},\omega) = H_i(\mathbf{x})\left(1-\int_\Omega W(\mathbf{x},\omega,\omega')f(\mathbf{x},\omega,\omega')(\omega'\cdot\mathbf{n})\mathrm{d}\omega'\right) + \int_\Omega \acute{C}_i(\mathbf{x},\omega')W(\mathbf{x},\omega,\omega')f(\mathbf{x},\omega,\omega')(\omega'\cdot\mathbf{n})\mathrm{d}\omega', \quad (1)$$

where $C_i(\mathbf{x},\omega)$ is the output coverage for object identifier i, $\mathbf{x}$ is the surface position, ω is the view direction, $H_i(\mathbf{x})$ is the hit of the object identifier: $H_i(\mathbf{x}) = 1$ if the identifier at $\mathbf{x}$ is i otherwise $H_i(\mathbf{x}) = 0$, ω' is the incoming direction, $\mathbf{n}$ is the surface normal, $f(\mathbf{x},\omega,\omega')$ is the BSDF, and $W(\mathbf{x},\omega,\omega')$ is a weighting function to reduce the amount of the passing through. $\acute{C}_i(\mathbf{x},\omega')$ is the incoming coverage for each identifier that is computed recursively. The first term of Eq.1 is the percentage of rays not passing though for the object identifier i, while the second term is the rest of the percentage for i. We solve this equation using a Monte Carlo integration simultaneously with the pixel color in path tracing as we wrote Eq. 1 in a similar form to the rendering equation. Since the number of identifiers can be large, we compute coverages only for hit identifiers during path tracing.

We can express the existing work that ignores glossy bounces as a special case of our form by setting a step function as a weighting function: $W(\mathbf{x},\omega,\omega') = 1$ if $f(\mathbf{x},\omega,\omega') = \infty$ due to a delta function (e.g., transparent or perfect specular surfaces) otherwise $W(\mathbf{x},\omega,\omega') = 0$. In this paper, we introduce a new continuous weighting function $W(\mathbf{x},\omega,\omega') \in [0,1]$ for glossy surfaces.

3.2. Weighting Function for Glossy Surfaces

The weighting function reduces the contribution of the incoming coverage $\acute{C}_i$ as details of reflected or refracted objects are lost by the BSDF. Blurring due to glossy surfaces does not depend only on the BSDF but also the distance. Therefore, we use a weighting function based on the spread of ray cones [ACB*21]. The growth of the cone spread is based on the surface curvature and the variance of the BSDF. We use a ratio with and without the BSDF variance at the bounce to measure the loss of object details for each bounce as shown in Fig. 2b.



(a) *Scattered rays due to BSDFs* **(b)** *Spread of our ray cones*

Figure 2: *(a) Scattering of rays in multiple bounces. The variance of rays increases due to the BSDF at every bounce. (b) Spread of our ray cones on a glossy surface. The red-dot cone simulates a perfect specular reflection case, while the solid-blue cone indicates the spread of our ray cone based on the variance of the BSDF.*

The variance of rays increases according to the product of a BSDF variance and the squared ray distance from the path vertex of the BSDF. In multiple bounces as illustrated in Fig. 2a, a ray variance at m-th bounce caused by a BSDF is given as $\nu_j^2(\sum_{k=j}^m t_k)^2$ where ν_j^2 is the BSDF variance at the j-th bounce, and t_k is the ray distance at the k-th bounce. Thus, the ray variance caused by all the BSDFs σ_m^2 is given by the total of them:

$$\sigma_m^2 = \sum_{j=0}^m \nu_j^2 \left(\sum_{k=j}^m t_k\right)^2. \quad (2)$$

However, this variance calculation is computationally expensive. Therefore, we first roughly approximate the square of the total distance with the total of squared distances as follows: $\sigma_m^2 \approx \sum_{j=0}^m \nu_j^2 \sum_{k=j}^m t_k^2$. Then, for ease of computation during path tracing, we equivalently rewrite it in the following recursive form:

$$\sigma_{m+1}^2 \approx \sigma_m^2 + t_m^2 \sum_{j=0}^m \nu_j^2. \quad (3)$$

Although the approximation error of σ_m^2 can increase if the bounce count m is high and BSDF variance at an earlier bounce ν_j is high, this error has little effect on the resulting coverage. This is because we reduce the coverage at subsequent bounces based on the ray variance at earlier bounces with less approximation errors. Unlike Akenine-Möller et al. [ACB*21], we calculate the BSDF variance ν_j^2 based on the PDF p_j of the BSDF to consider the directionally varying variance. Then, we represent this variance as the angle θ of our ray cone. Since the inverse of the PDF is a solid angle, the cone angle θ is given by the unit spherical cap: $\min(1/p_j, 2\pi) = 2\pi(1-\cos\theta)$. Thus, we obtain the tangent-space variance ν_j^2:

$$2\nu_j^2 = \tan^2\theta = \frac{1}{\left(1-\min\left(\frac{1}{2\pi p_j},1\right)\right)^2} - 1. \quad (4)$$

Therefore, the ratio of the cone-area spread due to the BSDF is

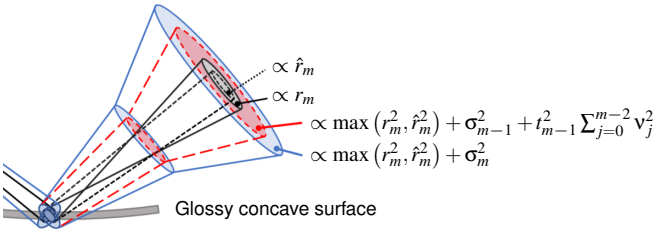


Figure 3: *Four ray cones used for our weighting function. Clamping the cone radius r_m^2 by $\hat{r}_m^2$ avoids artifacts caused by over shrinking of r_m^2 due to a concave surface.*

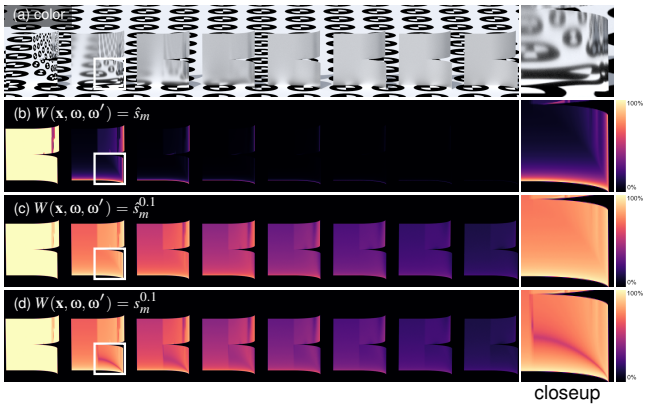


Figure 4: *(a) Curved planes with various roughnesses (roughness range: 0.0–0.7 with a 0.1 stride). (b–d) Visualization of indirect coverages for the floor using different weighting functions. A weighting function without radius clamping (d) produces artifacts on the concave surface, while the proposed method (c) does not.*

given as follows:

$$s_m^2 = \frac{r_m^2 + \sigma_{m-1}^2 + t_{m-1}^2 \sum_{j=0}^{m-2} v_j^2}{r_m^2 + \sigma_m^2}, \tag{5}$$

where r_m is the radius of the cone before spreading by the BSDF, and the numerator is proportional to the cone area that the previous BSDF is replaced with a perfectly specular surface. This s_m indicates a ratio of how cone radius is spread due to the glossy BSDF on the previous shading point. However, we observed that Eq. 5 produces noticeable artifacts when r_m is close to zero due to a negative cone-spread angle produced by concave surfaces. To alleviate these artifacts, we use another radius $\hat{r}_m = r_0 + \tan\theta_0 \sum_{j=0}^{m-1} t_j$ that is spread from the initial cone radius r_0 with the initial cone-spread angle θ_0 at a camera. We clamp the cone radius r_m by this $\hat{r}_m$ to avoid using over-shrunk r_m as follows:

$$\boxed{\hat{s}_m^2 = \frac{\max\left(r_m^2, \hat{r}_m^2\right) + \sigma_{m-1}^2 + t_{m-1}^2 \sum_{j=0}^{m-2} v_j^2}{\max\left(r_m^2, \hat{r}_m^2\right) + \sigma_m^2}}. \tag{6}$$

Fig. 3 shows the ray cones that we trace for our method. Fig. 4b visualizes the indirect coverage of the floor with $W(\mathbf{x}, \omega, \omega') = \hat{s}_m$. Although $\hat{s}_m$ represents the ratio of the blurred pixels due to the BSDF, it may not be perceptually proportional to object details for artists. Therefore, we use a tweakable map from $\hat{s}_m$ to the perceptual weight as follows:

$$\boxed{W(\mathbf{x}, \omega, \omega') = \hat{s}_m^{\alpha}}, \tag{7}$$

where α is a user-specified parameter to control the perceptual weight for artists. Fig. 4c shows the coverage for the reflection of the floor at the second bounce using this weighting function with our default parameter $\alpha = 0.1$. The area on the curved planes that has the less detailed reflection of the floor is assigned less weight and the stronger weights are kept against sharp reflection. Fig. 4d shows artifacts from Eq. 5 while they are avoided by Eq. 6.

3.3. Another Option for Coverage Computation

Our coverage computation shown in Eq. 1 reduces the coverage at the current fragment, and then redistributes the remaining coverage to the subsequent reflected/refracted fragments. However, artists may require a non-reduced coverage for opaque surfaces. Although this coverage can be computed by summing all the coverages of subsequent scattered fragments, we do not store the path of fragments (represented with a tree structure) because of the storage cost and image formats (e.g., Cryptomatte does not support the path of fragments). For the case where artists require the non-reduced coverage for opaque surfaces, we provide an option that reduces the coverage only for transparent surfaces as follows:

$$C_i(\mathbf{x}, \omega) = H_i(\mathbf{x})\left(1 - T(\mathbf{x})\right) + \int_\Omega \acute{C}_i(\mathbf{x}, \omega') W(\mathbf{x}, \omega, \omega') f(\mathbf{x}, \omega, \omega')(\omega' \cdot \mathbf{n}) d\omega', \tag{8}$$

where $T(\mathbf{x})$ is the transparency of the BSDF. Unlike Eq. 1, the sum of all the coverages in a pixel can exceed 100% in this form. Therefore, artists have to do image manipulation considering this limitation. Since $1 - T(\mathbf{x})$ contains all the subsequent coverages after the reflection/refraction, $H_i(\mathbf{x})$ in this form should be zero if the ID i is found in previous bounces. This can be done by keeping IDs found in all previous bounces, however, this is computationally expensive. Instead, we set $H_i(\mathbf{x})$ zero only when the previous bounce has the same ID as a cheap practical approximation to handle interreflections on a single object in our implementation.

4. Our ID–Coverage Update on the GPU

GPU path tracing can assign multiple threads for different samples in a pixel. To update the pair of an ID and coverage for each pixel, we use a lock-free algorithm that avoids data races by using a compare and swap (CAS) instruction and linear probing. Algorithm 1 shows our coverage accumulation algorithm using CAS. For this CAS instruction, an ID–coverage pair is packed as a 32-bit or 64-bit variable. Although the memory usage for multiple fragments in complex scenes is typically unknown before the rendering process, we use a fixed per-pixel storage size for the GPU efficiency, similar to the k-buffer [CICS05]. Accordingly, our implementation cannot store all the fragments when the fragment count is larger than the storage size. In this case, we prioritize directly visible fragments, since they are more visually noticeable than indirectly visible frag-

Algorithm 1: Update of a pair of ID and coverage on the GPU. **Inputs :** *ID*: an object identifier at a shading point. *contribution*: a value adding to the coverage of the object identifier. This value is set to negative when the object is indirectly visible. *idCoverages*: array for ID–coverage pairs. *N*: array size of the idCoverages.

```
function Update (ID, contribution, idCoverages, N)
    base ← Hash (ID) %N;
    k ← 0;
    minIdx ← −1;
    minIdCvg ← {NULL, −∞};
    while k < N do
        index ← (base + k)%N;
        current ← idCoverages[index];
        if current.id = ID ∪ current.id = NULL then
            sign ← (current.coverage > 0 ∪ contribution > 0) ? 1 : -1;
            newCoverage ← |contribution| + |current.coverage|;
            new ← {ID, sign ∗ newCoverage};
            old ← CAS (idCoverages[index], current, new) ;
            if old = current then break ;
            else k ← k − 1 ;
        else if current.coverage < 0 ∩ current.coverage > minIdCvg.coverage then
            minIdx ← k;
            minIdCvg ← current;
        end
        if contribution > 0 ∩ k + 1 = N ∩ minIdx ≥ 0 then
            new ← {ID, contribution};
            old ← CAS (idCoverages[minIdx], minIdCvg, new) ;
            if old ≠ minIdCvg then
                minIdx ← −1;
                minIdCvg ← {NULL, −∞};
                k ← −1;
            end
        end
        k ← k + 1;
    end
end
```

ments. For this prioritization, we store whether the fragment is directly visible or indirectly visible into the sign bit of the coverage. When the per-pixel storage is full, we evict an indirectly visible fragment with the smallest coverage. We use this approach for the simplicity and less memory consumption. Our approach, however, cannot prioritize fragments based on the true coverage because the stored coverage has variance due to Monte Carlo estimation. The bias due to the eviction of fragments may be reduced by using a stochastic approach such as reservoir sampling [Cha82]. We would like to study such stochastic update in the future.

5. Results

We implemented an existing approach [FJ15] focusing only on direct visibility and our approach extending it to glossy surfaces using OpenCL™. All the images in this paper are rendered using an AMD Radeon™ Pro W6800 GPU. Fig. 1 is the visualization of coverages and an example of re-coloring. The indirect visibility of several objects on the water surface was used for re-coloring in the right image. Fig. 5b shows a matte of plants with the existing approach while 5c shows a matte of the plants with our approach. Fig. 5b contains only direct visibility of the plants but 5c contains indirect visibility by complex indirect light transport effects in addition. Fig. 5d shows an image by hue adjusting against 5a by using 5c which is not possible with capturing only from direct visibility. Fig. 6 visualizes object coverages computed by our approach. The overhead of our method with storing fragments as Cryptomatte is less than 11% for these test scenes as shown in Fig. 6c.

6. Conclusions and Future Work

We have presented multi-fragment rendering with a weighting function considering the diffusion of rays based on ray cones. This makes it possible to capture indirectly visible fragments in multiple bounces as shown in our experimental results. We also presented an update method for a fixed-size multi-fragment storage on the GPU alleviating noticeable loss of fragments. Our current implementation stores only object identifiers unlike typical A-buffers that store an additional information such as depth. Thus, the granularity of our matte generation is limited to object identifiers. Our implementation does not separate the matte with depth and bounce types. For future work, we would like to extend the granularity of matte by encoding an additional information such as whether reflection or refraction into the ID. Although this approach does not require an additional element such as a depth channel for the image format, it can increase the fragment count in the storage and may cause more evictions of fragments. This is because fragments with different IDs are separately stored. Studies to maximize the benefits of these additional information under limited storage size are left for future work.

Acknowledgments

We would like to thank Morgenrot Inc. for creating the scene of Fig. 5. We also thank Yohsuke Nakano at Morgenrot Inc. for valuable comments. The scenes of the second row and the third row in Fig. 6 were created by ACCA software. We are grateful to let us to use their software for our research. We sincerely thank Pieterjan Bartels at Advanced Micro Devices, Inc. and Oleksandr Kupriyanchuk at Luxoft for proofreading and constructive suggestions. Also, many thanks to Li-Yi Wei for valuable advice to revise the paper.

References

[ACB*21] AKENINE-MÖLLER, T., CRASSIN, C., BOKSANSKY, J., et al. "Improved shader and texture level of detail using ray cones." *J. Comput. Graph. Tech.* 10.1 (2021), 1–24 2.

[Car84] CARPENTER, L. "The A-Buffer, an Antialiased Hidden Surface Method". *SIGGRAPH Comput. Graph.* 18.3 (1984), 103–108 2.

[Cha82] CHAO, M. T. "A General Purpose Unequal Probability Sampling Plan". *Biometrika* 69.3 (1982), 653–656 4.

[CICS05] CALLAHAN, S.P., IKITS, M., COMBA, J.L.D., and SILVA, C.T. "Hardware-assisted visibility sorting for unstructured volume rendering". *IEEE Trans. Vis. Comput. Graph.* 11.3 (2005), 285–295 3.

[FJ15] FRIEDMAN, J. and JONES, A. C. "Fully Automatic ID Mattes with Support for Motion Blur and Transparency". *SIGGRAPH '15 Posters*. 2015 2, 4, 5.

[Psy15] PSYOP. *Cryptomatte*. https://github.com/Psyop/Cryptomatte. (Retrieved Dec. 10, 2021). 2015 1.

[ST90] SAITO, T. and TAKAHASHI, T. "Comprehensible Rendering of 3-D Shapes". *SIGGRAPH Comput. Graph.* 24.4 (1990), 197–206 2.

[VVP20] VASILAKIS, A. A., VARDIS, K., and PAPAIOANNOU, G. "A Survey of Multifragment Rendering". *Comput. Graph. Forum* 39.2 (2020), 623–642 1, 2.

(a) *Rendered image* **(b)** *Previous matte [FJ15]* **(c)** *Our matte* **(d)** *Re-colored image using our matte*

Figure 5: *Re-coloring of the rendered image using a matte in a postprocess image editing software. Some plants are hidden by the glass object in the matte generated considering only direct visibility (b). On the other hand, our matte (c) takes into account indirect visibility reflected and refracted from the glass object. Using our matte, we can easily change the color of plants (d) after rendering.*

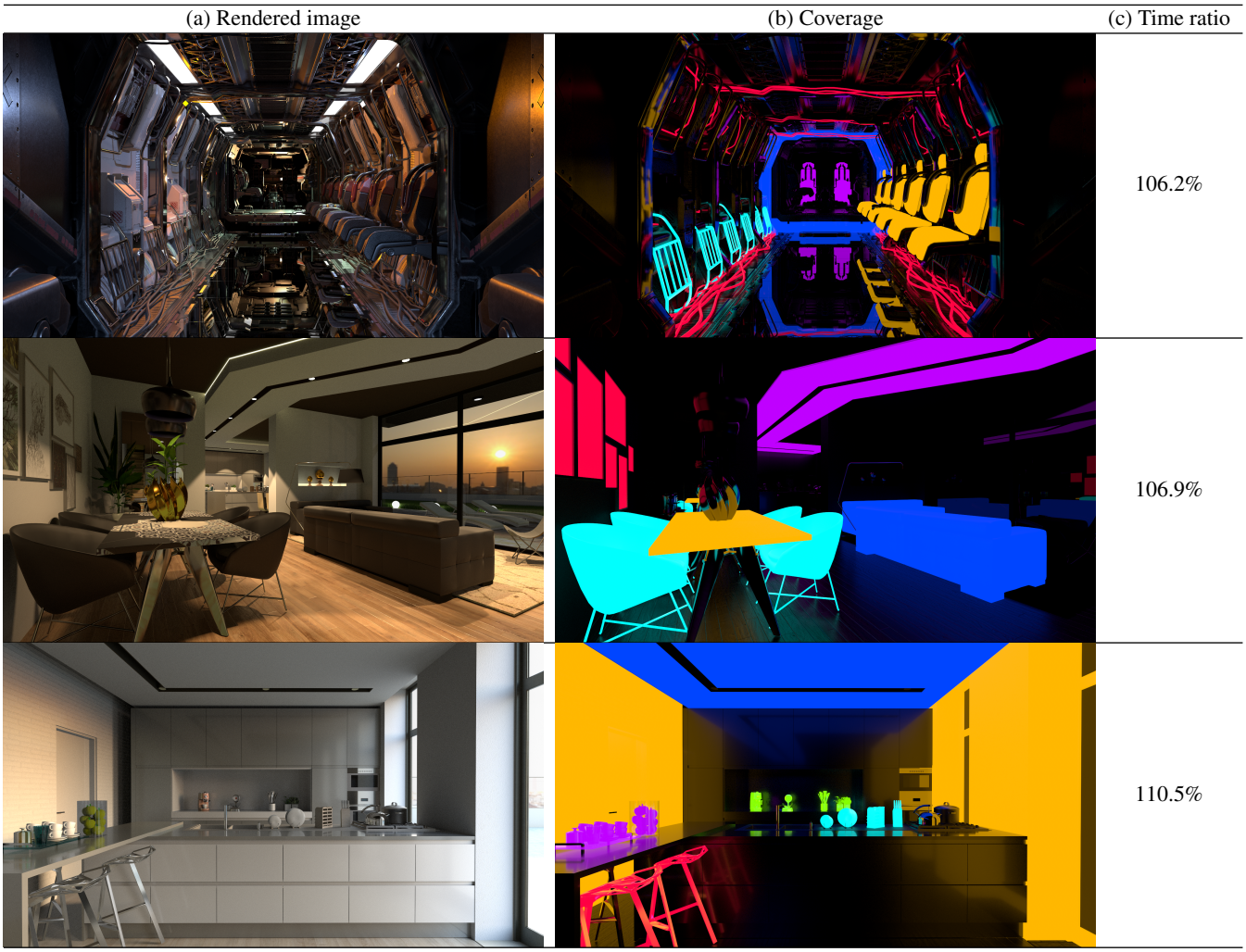


Figure 6: *Rendered images (a) and coverage visualizations (b) of our approach (1920×1080 pixels, 12 elements per pixel, 1024 samples per pixel). The right most column (c) is the ratio of the rendering time with and without Cryptomatte generation.*