Multi-Resolution Geometric Representation using Bounding Volume Hierarchy for Ray Tracing

Sho Ikeda
Advanced Micro Devices, Inc.
Japan

Paritosh Kulkarni
Advanced Micro Devices, Inc.
Canada

Takahiro Harada
Advanced Micro Devices, Inc.
USA

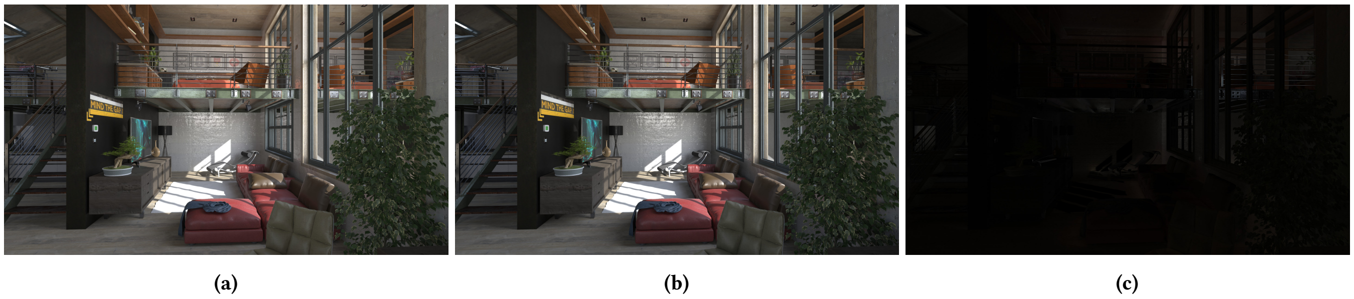


Figure 1: (a) The reference path traced image of loft scene. (b) Rendered with the proposed method. (c) Difference between (a) and (b) shows the error from the method. Rendering time and ray casting time were reduced by 6 % and 12 %, respectively, using our method.

ABSTRACT

This paper presents a multi-resolution geometric representation using a bounding volume hierarchy (BVH). Our method makes it possible to lower the computational cost of ray tracing by using an approximated geometry that is taken from the acceleration data structure we usually build for ray tracing, i.e., the BVH. Our method implements a level of detail (LOD) for rendering by adding small logic to the existing BVH traversal algorithm, and does not require any precomputation, nor does it add any additional data for geometric representation. At the same time, simplifying the geometry is not sufficient for approximation of non-shadow rays. Therefore, we also propose methods that allow us to use a geometric approximation for all ray types. Specifically, we propose stochastic material sampling with a small memory overhead to realize material blending, a method that enables us to use the proposed approximation in a path tracing algorithm as shown in the paper.

KEYWORDS

Ray tracing, Global illumination

1 INTRODUCTION

Monte Carlo ray tracing is a simple and robust rendering algorithm that has been studied for years [Kajiya 1986]. Ray casting is the core component of ray tracing, which scales better in terms of the number of primitives in the scene than rasterization which has linear complexity. However, even with the logarithmic complexity in ray tracing, the computational cost is still high for real-time applications. Therefore, developers have been attempting to minimize the number of rays to be cast, and rely heavily on denoising in video games [Haines and Akenine-Möller 2019]. Hence, the use of ray tracing is still very limited in today's video games.

This paper describes the following two major contributions. The first is the multi-resolution geometric representation using axis-aligned bounding boxes (AABB) in a bounding volume hierarchy (BVH). The second is the stochastic material sampling on any LOD node. The multi-resolution geometric representation reduces the computational complexity in ray casting without requiring any additional precomputation or storage, and the geometric LOD is implemented by adding a small change to an existing BVH traversal algorithm. Although the use of AABBs in BVH for geometric representation is a simple idea, it can be only used for occlusion rays as information needed for other rays in global illumination algorithm is not there. Specifically, we need to evaluate materials on a coarse representation of the geometry to use the geometric approximation for non-occlusion rays. This paper proposes a solution for this, the stochastic material sampling for material evaluation at any LOD level of the geometry. The algorithm works for complex materials with a shading network and makes it possible to filter materials on the geometries the LOD represents. The memory overhead is small for this method. We only add two integers for each node of a BVH. As we show in this paper, applying the geometric LOD using the proposed method for ray casting is very straightforward and can be added to any ray tracing engine by modifying its BVH traversal logic. As stochastic material sampling is also simple and does not put any restrictions on the used material, it can be easily integrated into any existing renderer supporting a complex material system as shown in Fig. 1. At last, we evaluate the performance quantitatively.

email: { Sho.Ikeda, Paritosh.Kulkarni, Takahiro.Harada }@amd.com
Advanced Micro Devices, Inc. Technical Report No. 22-02-f322, February 28, 2022.

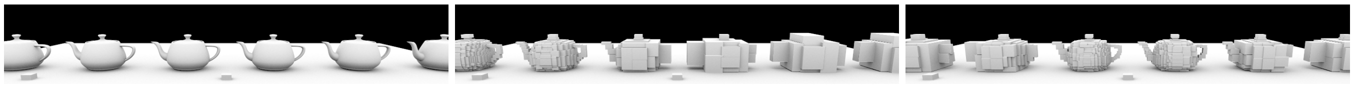


Figure 2: Visualization of geometric representation from diffrent locations. Left: Polygonal representation, Center and right: The ray origin is defined at the cube on the floor on the left and center, respectively.

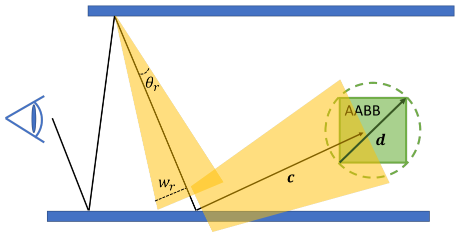


Figure 3: 2D illustration of ray cone and AABB.

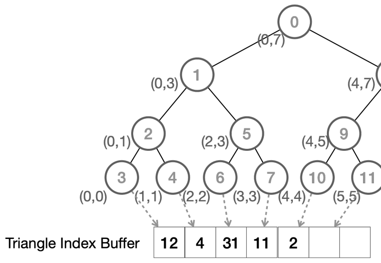


Figure 4: Data stored in the BVH for stochastic material sampling.

2 MULTI-RESOLUTION GEOMETRY REPRESENTATION USING A BOUNDING VOLUME HIERARCHY

2.1 Geometric Representation

Instead of explicitly modeling LOD meshes and switching them as done in [Won-Jong Lee and Vaidyanathan 2019], we utilize the existing data structure for ray tracing, which is the BVH. Each node in a BVH stores its AABB, which encloses all the triangles that we can find in its descendants. The AABB is built such that it conservatively bounds the geometry, and we use AABBs as a coarse approximation of the geometry similar to the work by Lacewell [Lacewell 2008] who uses it for shadow filtering. This gives us a multi-resolution representation of the geometry without requiring any additional precomputation or memory overhead. When a more accurate representation is necessary, we can descend the tree and use the approximation at a lower level of the tree. We can use the AABB from a node close to the root if we want a rough approximation, such as the case where the geometry is far from a shading point.

Using the AABB in a BVH as the geometric representation has some useful aspects. First, it provides a conservative bound, therefore does not introduce any holes. Secondly, a single approximation level does not need to be defined for the entire mesh. Instead, we can have different levels of representation in a single mesh as shown in Fig. 2, where it transitions from the highest resolution representation (polygons), to a coarse representation as the distance from the point of interest increases. Finally, it is also possible to select the level for each ray at runtime, so as to give us flexibility on the level selection, which is not possible in the existing method (Fig. 2) [Lloyd et al. 2020].

2.2 Level of Detail Selection in Ray Tracing

During regular ray traversal, when a ray intersects an AABB, the ray continues traversal within that AABB until it reaches a leaf node. With our geometric approximation, we first check if the AABB of a node is good enough to be used as the geometric proxy of all the triangles covered by the node. When we have such an AABB, the ray instead uses the intersection as a hit and stops traversing its descendants.

We compute a ray cone tracked with a ray that grows with spread angle θ_r (Fig. 3). In order to alleviate the geometric error caused by the approximation, the θ_r is dynamically changed by the number of ray bounces and materials at shading points which we will discuss the details in Sec. 3.2. This ray cone is different from the ray cone used for texture filtering since it does not spread the angle from the camera but from the first diffuse vertex. Also as the spreading logic is different as we discuss in Sec. 3.2 from the ray cone used for texture filtering, another ray cone tailored for the geometric approximation is traced. When the ray intersects an AABB, we check if the virtual sphere which encloses the AABB is entirely overlapped by the ray cone by computing a condition $|\mathbf{d}| < 2\,|\mathbf{c}|\,tan\,(\theta_r) + 2w_r$, where $\mathbf{d}$ is the diagonal vector of the AABB, $\mathbf{c}$ is the vector from the ray origin to the center of the AABB and w_r is the width of the ray cone of previous path segment. If the condition is true, the AABB is used as the geometric proxy.

3 EXTENSION TO PATH TRACING

Although finding the intersection point can be used for occlusion rays, it is not sufficient for use in path tracing, which was the limitation in [Yoon et al. 2006]. In this section, we propose methods required to use the geometric approximation in a path tracing algorithm.

3.1 Stochastic Material Sampling

The most important information we need to provide is the material information for shading. Our approach does not make any assumptions about the material on the geometry we simplify, i.e., it can be used for a material with a complex shading network.

An important point to consider is how to obtain the material information at a geometric proxy. Storing the material itself at a node is not practical in production rendering, where blending of materials is common. Blending cannot be done easily for cases where the

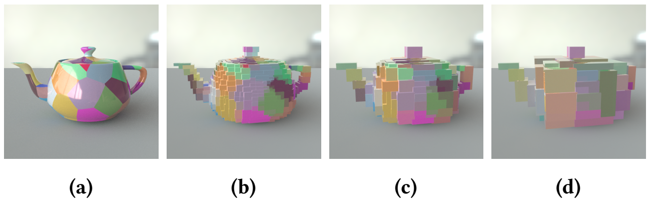


Figure 5: Example of the geometric approximation and stochastic material sampling. (a) is the original polygonal representation. (b), (c), (d) are approximation with 2, 4 and 8 degrees, respectively.

BSDF is used in descendants with a large variety of different materials, or where complex shading networks are involved. Another problem with this approach is the memory overhead required to store BSDFs with multiple parameters. We propose the stochastic material sampling requiring only two integer values stored in each node, which can be precomputed by doing tree traversal once. We store the range of the triangle indices of the node's descendants in a node of the BVH, as shown in Fig. 4. Once we have found the intersecting node for the ray, we sample the triangles from the range using uniform sampling assuming the triangles are uniformly distributed in the direction of the ray. After one triangle is selected, we use its material for shading. This approach is similar to the work by Cook et al., [Cook et al. 2007], but our sampling is done at runtime for each ray.

We also need to obtain the texture coordinates for the hit point. We again rely on stochastic sampling, i.e., we sample a barycentric coordinate in the triangle we selected, and use it to interpolate the texture coordinates of the vertices. For the shading normal, we could also interpolate it as we do for texture coordinates, however, we found out that using the geometric normal of the AABB is sufficient. A scene with a procedural material network and its simplification are shown in Fig. 5, where we can see that the materials are blended together as we proceed to a coarser representation.

3.2 Dynamic Spread Angle

Geometric error caused by the approximation used for direct illumination introduces a significant error to the rendered images. In such a case, the approximation is not appropriate. On the other hand, further bounces do not contribute much to the rendered images, but still have noticeable computation time due to incoherent rays. For such paths, we can use the approximation to accelerate ray casting. Considering those, we calculate the spread angle θ_r dynamically for each ray. We disable ray cone ($\theta_r = 0$) with camera ray and occlusion ray for direct illumination, otherwise we use $\theta_r = max\left(H\left(b_d - 2\right)T, H\left(b_d - 1\right)\theta_\delta\right)$ where $H\left(x\right)$ is heaviside step function $H\left(x\right) = 1$ if $0 \leq x$ and $H\left(x\right) = 0$ if $x < 0$, b_d is the number of diffuse bounces in a path, T is user specified spread angle and θ_δ is small spread angle. We use $\theta_\delta = 3^\circ$ in this paper. Although this logic increases θ_r as the ray bounces more on a diffuse surface, it does not change the angle at a reflection on a specular surface.

3.3 Implementation Details

When the geometric approximation is used on area light, it can cause noticeable error. Specifically these error arise due to geometric approximation changing surface area of the area light. To avoid this issue, we track emissive materials when building a BVH and set a flag on BVH nodes containing area light in its subtree.BVH nodes having this special flag would skip the geometric proxy check, and thus the area lights are never approximated.

4 HARDWARE RAY TRACING EXTENSION

The geometric approximation is simple enough so it could be implemented in the existing hardware ray tracing instruction. We propose an extension to hardware ray tracing instruction *image_bvh_intersect_ray* [Advanced Micro Devices, Inc. 2020]. First, we think about implementing the geometric approximation into the ray traversal using existing *image_bvh_intersect_ray*. Currently *image_bvh_intersect_ray* returns sorted child node indices when input node is internal node, and we do not have enough information for determining if the child nodes can be approximated. In order to determine if we can apply the approximation or not which we explained in Sec. 2.2, we need to obtain ray hit distance for AABB, centroid and diagonal length of AABB. We also need ray hit face of AABB for calculating geometric normal for shading. However computing the geometric information outside of *image_bvh_intersect_ray* is not efficient because of redundant computation and access to the AABB data. Hence, we propose an extension to *image_bvh_intersect_ray* to determine if child nodes can be approximated.

Our proposed instruction requires additional values and returns values for us to decide whether we can use the approximation or not. We pass the parameter θ_r for the approximation. The instruction returns values which are ray hit distances, flags to tell if the hit distance is valid or not and hit face of AABB. These can be encoded into 3 bits. To keep the data returned from the instruction small, we encoded the hit distance into 29 bits so we can pack all the information needed for a single AABB into 32 bits. Since the hit distances for AABBs are computed in the instruction, we only need to add the conditional check if we can apply the approximation or not after the hit distance computation of the instruction. We show the pseudo code of geometric approximation using the extended instruction in Listing 1. We call *image_bvh_intersect_ray* with *approxParams* which contains the parameters for geometric approximation, and get the result value *approxResult* which is encoded hit distance, flag and hit face (line 7 and 8). After the instruction returns, we check the flag of the return value if we can apply the approximation or not (line 21). The only additional logic for the user's code is between line 21 to 26.

```
1  MinHit minHit; // Store the shortest ray hit from ray origin
2  NodeStack stack; // Stack of BVH nodes which a ray will traverse
3  push(rootNode, stack);
4  while(0 < StackSize(stack))
5  {
6    node = pop(stack);
7    float4 approxResult; // encoded value of hit distance, a flag to
         tell if the hit value is valid and hit face
8    uint4 result = image_bvh_intersect_ray(node, ray, approxParams,
        &approxResult);
9    if(isLeafNode(node)) // Leaf node case
10   {
11     // process triangles
12     updateMinHit(...); // Update minHit value with shortest hit
```

```
  }
  else // Internal node case
  {
    for(int i = 0; i < 4; i++)
    {
      uint hitNode = result[i];
      if(hasHit(hitNode))
      {
        if(getFlag(approxResult[i])) // Check if the underlying
      hit distance is valid
        {
          float4 geometricNormal = getFaceNormal(approxResult[i]);
          updateMinHit(getHitDistance(approxResult[i]),
      geometricNormal, ...);
        }
        else
        {
          push(hitNode, stack);
        }
      }
    }
  }
}
```

Listing 1: Geometric approximation pseudo code using extended hardware ray tracing instruction. The extension of *image_bvh_intersect_ray* takes the parameters of geometric approximation and check the condition if we can apply the approximation or not after the hit distance computation of the instruction.

5 RESULTS

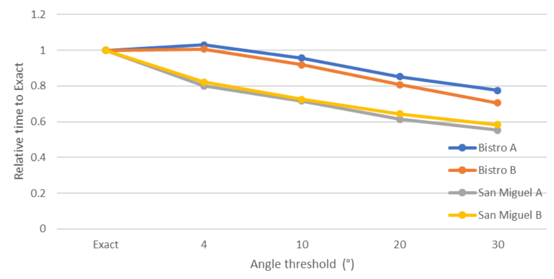


Figure 6: Comparison of relative rendering time with different spread angle T for Bistro A, B, San Miguel A, B.

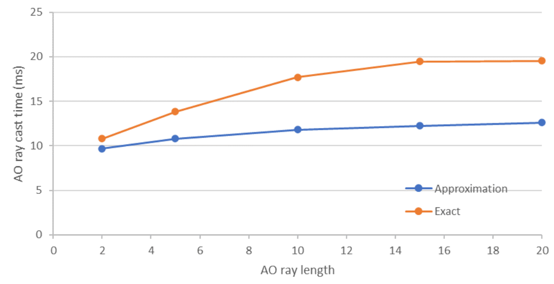


Figure 7: Comparison of AO ray cast time with different AO ray length for Bistro A.

The method is implemented using OpenCL™. All the tests were executed at 1920×1080 resolution on a machine with an AMD Ryzen™9 5950X CPU and an AMD Radeon™RX 6800XT GPU.

We first measured the ray casting time for ambient occlusion (AO) rays for two scenes, each with two different views (A and B). Obviously, as we use a larger angle for the spread angle T, the approximation gets rough. It can be seen in Fig. 8 where (c) is darker than (a) while we can see a drastic reduction in the number of nodes in the BVH it traverses (up to 31% if we compare left and right in Fig. 8). Fig. 6 shows the relative time for AO ray casting. The improvement is smaller when the spread angle T is smaller, but we can see that we obtain an approximate 40% speed up in the best case. The improvement also depends on the AO ray length which is shown in Fig. 7. We can observe that our method causes only a negligible overhead in comparison with the exact solution.

The proposed method was integrated into an existing path tracer to evaluate the improvement in rendering time. First, we evaluated how our approximation affects rendering in path tracer with simple scene (Fig. 9). The error is concentrated in the region of the reflection on the sphere which is deeper path segment in the simple scene. In such path, a coarser representation of the sphere is used and it causes different occlusion computation from the original sphere. Next, we evaluated the proposed method in more practical scenes as shown in Table 1. We can see some errors in the rendered images which vary depending on the scene. The Steam Edo scene has the minimum visual error as the indirect illumination in the scene does not contribute so much. On the other hand, the Room, Hangar scenes have larger error on some part of the images. These come from multiple specular reflections and refraction which cannot be represented well with our approximation. Rendering time and ray casting time have improved up to 14 % and 12 %, respectively. As the ray traverses deeper, we observed that the improvement is more significant (about 2×) since more rays use the approximation. However, the improvement in the Room scene is not as good as in other scenes because the execution was mostly bottle-necked by other computations such as shader execution. Therefore, improvement in the render time is bigger for a scene where there are more longer paths. We used $T = 30°$ for the spread angle.

6 CONCLUSION

We have showed that the proposed method allows to use an approximated geometric representation not only for occlusion rays but also for rays in path tracing thanks to the stochastic material sampling from which we obtain all the information we need to extend a path. We also evaluated the visual error and rendering time improvement with the method under complex material and lighting scenarios.

Future work includes extension to transmissive surfaces, use of hardware ray tracing, and application to the animation. There is a room for improving the condition of the approximation application where it produces a large error (for example a light is coming from the hole of a torus and the torus is approximated as a box without the hole). Another point of improvement would be for the stochastic material sampling where uniform sampling is not always a good sampling strategy when the distribution of the triangles is not uniform. Taking triangle area and directionality into consideration

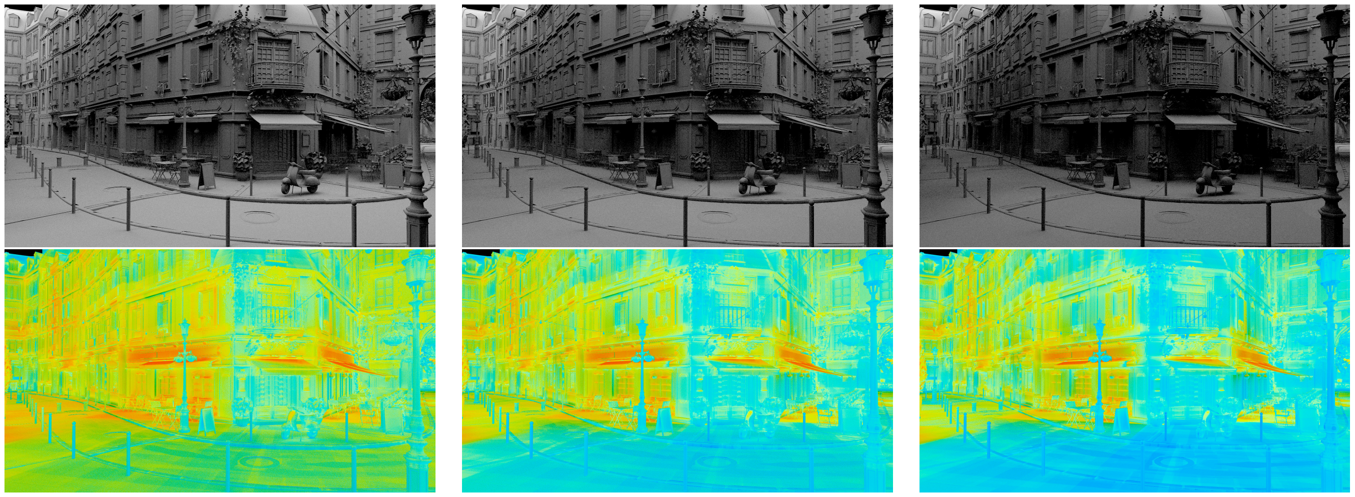


Figure 8: Bistro A, rendering images and visualizations of number of nodes traversed in the AO ray cast. Left is exact. Center and right are approximation with T = 10 and 20 degrees respectively. Bottom row shows the number of nodes each AO ray traverses to find intersection. Blue (min) is 0 and red (max) is 400. Average number of nodes each AO ray traverses in pixels are 317, 240 and 218 respectively.

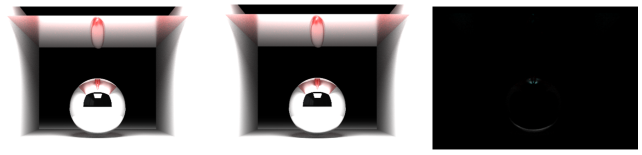

(a) exact **(b) with our method** **(c) absolute difference**

Figure 9: Simple scene showing the difference our method produces.

with low memory overhead is another possible extension left for future work.

ACKNOWLEDGEMENT

The loft scene was created by ACCA software. The Junk Shop was created by Alex Treviño. Steam Edo was created by Shunsuke Nakajo. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

Advanced Micro Devices, Inc. 2020. "RDNA 2" Instruction Set Architecture Reference Guide. https://developer.amd.com/wp-content/resources/RDNA2_Shader_ISA_November2020.pdf.

Robert L. Cook, John Halstead, Maxwell Planck, and David Ryu. 2007. Stochastic simplification of aggregate detail. ACM Trans. Graph. 26, 3 (2007), 79. https://doi.org/10.1145/1276377.1276476

Eric Haines and Tomas Akenine-Möller (Eds.). 2019. Ray Tracing Gems. Apress. http://raytracinggems.com.

James T. Kajiya. 1986. The rendering equation. In Computer Graphics. 143–150.

Dylan Lacewell. 2008. Raytracing prefiltered occlusion for aggregate geometry. In Proceedings of the IEEE Symposium on Interactive Raytracing.

Brandon Lloyd, Oliver Klehm, and Martin Stich. 2020. Implementing Stochastic Levels of Detail with Microsoft DirectX Raytracing. https://developer.nvidia.com/blog/implementing-stochastic-lod-with-microsoft-dxr/.

Gabor Liktor Won-Jong Lee and Karthik Vaidyanathan. 2019. Flexible Ray Traversal with an Extended Programming Model. In Proceedings of ACM SIGGRAPH Asia 2019, Technical Brief. 17–20.

Sung-Eui Yoon, Christian Lauterbach, and Dinesh Manocha. 2006. R-LODs: fast LOD-based ray tracing of massive models. The Visual Computer 22, 9 (2006), 772–784.

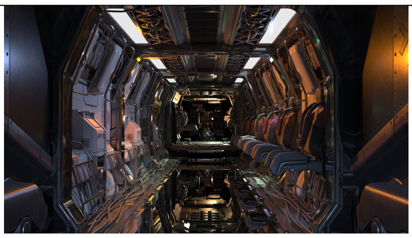
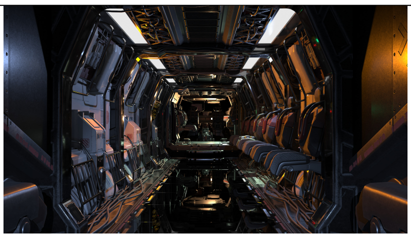
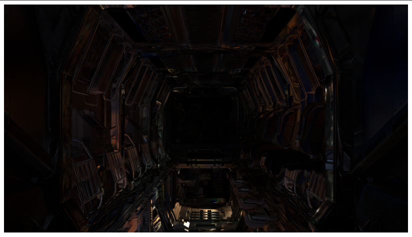
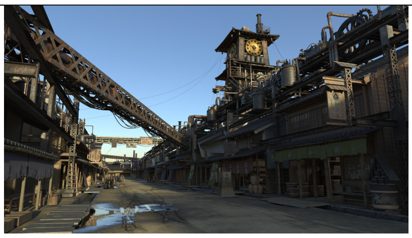
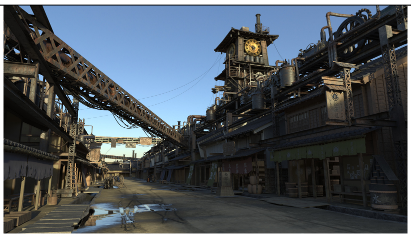
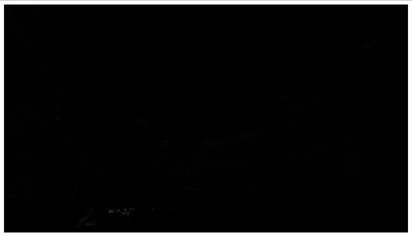
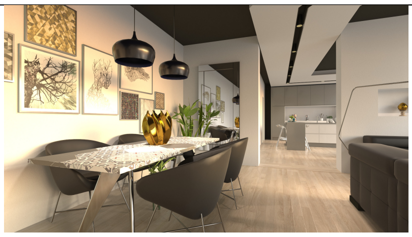
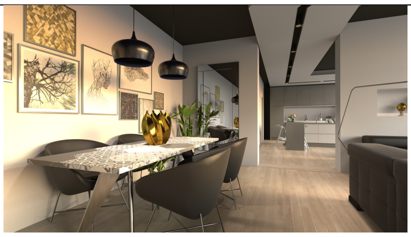
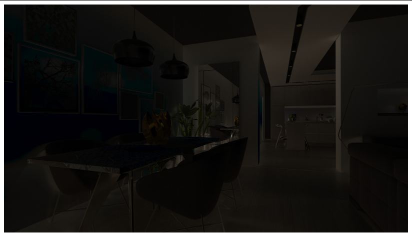
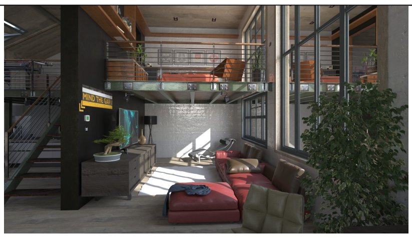
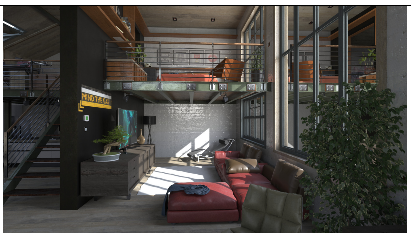
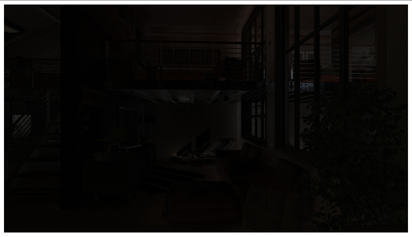
	Exact	Approximation	Difference
Hangar	 1799.17 74.08	 1588.66 67.57	
Steam Edo	 2116.12 84.78	 1812.74 82.97	
Room	 501.19 160.37	 489.72 152.76	
Loft	 448.93 163.75	 421.49 144.49	

Table 1: Comparison of images rendered using path tracing and modified path tracing with our approximation. Top number in the first and second columns is rendering time for one sample per pixel (spp) in milliseconds. Bottom number is a sum of all ray casting times for one spp in milliseconds.