CS 184 Assignment 3

Hector Li

August 2025

0 Overview

Logistical note: The PDF of this write-up and the original png files used in this document are all available on the website https://sigmundr6.github.io.

AI Usage Acknowledgment: I have used Google Gemini 2.5 Pro and GPT-4o for some concept clarifications on path tracing and high-level debugging guidelines for global illumination Section. The algorithms used in this assignments are eventually completed on my own.

In this assignment, we are implementing a basic path tracer running on CPU cores and achieves the basic functionalities such as direct illumination, global lumination, and adaptive sampling mainly on materials with diffuse bidirectional scattering distribution functions (BSDF). This path tracer simulates the light transport patterns under the assumption that light is purely geometric without the wave-like properties.

In Section 1, we will implement the fundamental tools of the ray-tracing pipeline, such as generating camera rays for each pixel and how they intersect with primitives such as triangles and spheres. Since the primitives to render in our scenes are sparse, we will implement the algorithm of Bounding Volume Hierarchy in Section 2 to reduce the computation of checking intersections. We construct a tree of bounding boxes and only check the ray intersections at the leaf nodes to filter out rays guaranteed not to not hit. In Section 3, we consider the most basic ray-tracing problem where we only consider the direct lighting, namely the light source (zero bounce) and the first direct bounce off the object (first bounce). By comparing the uniform hemisphere sampling and importance sampling, we explore how different sampling methods could be used to reduce noise. In Section 4, we trace the light transport further along its indirect bounces. We first use field `max_ray_depth` to generate biased pictures. However, since Monte Carlo Integration could not predict future light transports or the convergence of the recursion, we will implement the Russian Roulette method to generate unbiased pictures. Finally, in Section 5, we will introduce adaptive sampling using the statistical concept of confidence interval to reduce the number of light samples generated on pixels that have little variance on each ray to allow us sampling regions with high variances with more rays.

The most immediate impression of this assignment is how concept heavy it is. The bugs or irregular behaviors of the program are usually caused by

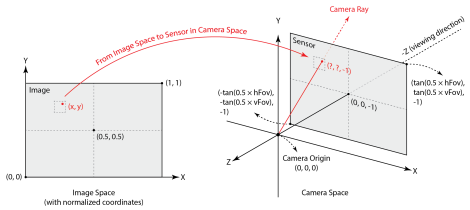

Figure 1: Camera model.

subtle but critical logical errors or edge cases that take a long time to discover and think over. The part about global illumination particularly demands a clear understanding of how path tracing works to fine-tune the many intricate components of the recursion. In addition, the pictures in this assignment take much longer to render than those in prior assignments. Without offloading the computation to GPU, even the most basic path tracing tasks on diffuse BSDF materials can take a considerable time. This rendering time discrepancy reminds us that what is simple in concept such as geometric light transport and diffuse BSDF can be computationally heavy.

1 Ray Generation and Scene Intersection

Unlike rasterization that projects three-dimensional geometry via triangles or quads onto the screen space, we send out sampling rays from every pixel in ray-tracing, simulate the light transport, and trace their paths and luminosity through the scene.

To effectively generate sampling rays, we first consider the simple model where the camera lies at the origin $(0, 0, 0)$ facing an image place at $z = -1$ parallel to the xy-plane. In our `Camera::generate_ray` function, given a coordinate (x, y) in the normalized image space, we want to construct a ray pointing from the origin to that point on the image in the world coordinate space. Note this process involves two changes of basis, both of which can be achieved by linear operations or matrix multiplications. In Figure 1, we note that the width of the image is `2 * std::tan(hFov / 180.0 * M_PI * 0.5)` and the height is `2 * std::tan(vFov / 180.0 * M_PI * 0.5)`, so in the coordinate system of the camera space, the coordinates of the point on the image space can be represented as `width * (x - 0.5)`, `height * (y - 0.5)` and `-1`. Therefore, we can compute the position of this ray in the world space by left multiplying the `c2w` matrix and the `pos` field is the place where the camera is placed. We also use the `unit` function to get the unit vector, since the direction vectors of rays are normalized. In addition, we do not want to consider the light transport patterns too close to the camera or too far away so we change the `min_t` and `max_t` fields to `nClip` and `fClip` to clip the ray between two planes.

With the camera model set up, we can consider our framebuffer to be the image placed at $z = -1$ location. Unlike rasterization where $(0, 0)$ is the top

left corner, in ray-tracing we use the normal Cartesian coordinate system. For each pixel at the place (x, y), we want to generate `ns_aa` sampling rays. Rather than sampling subpixels in a uniform grid like in supersampling, we use the Mersenne twister method provided by the `gridSampler` field to generate uniform random numbers in $[0, 1)$ for the x and y coordinates within this pixel. Then we can normalize the (x, y) coordinates of each point on the image by dividing them by `sampleBuffer.w` and `sampleBuffer.h` and use the camera model we discussed to construct sampling rays with their position based on the world space. With each sampling ray `r`, we accumulate the radiance with `est_radiance_global_illumination(r)` normalized by the sample number `ns_aa`. Then we update the pixel at (x, y) location on the sample buffer with `update_pixel(c, x, y)` and record the number of samples taken at this pixel in the `sampleCountBuffer` vector.

Since most of our physical objects are still modelled by triangles, we need to implement a method to compute ray intersection with a triangle efficiently. For a triangle with the vertices $P_1, P_2, P_3 \in \mathbb{R}^3$ in the counterclockwise orientation, we can compute its normal vector $\mathbf{n} = (P_2 - P_1) \times (P_3 - P_1)$. We then compute the dot product of this normal vector $\mathbf{n}$ with the directional vector $\mathbf{d}$ of the ray $\mathbf{o} + \mathbf{d}t$ for $\mathbf{o}, \mathbf{d} \in \mathbb{R}^3$ and $t \in \mathbb{R}$. If the absolute value of the dot product is less than `std::numeric_limits<double>::epsilon()` (considering the errors of float point operations), we claim that the ray is parallel to the triangle, so there is no intersection (if the ray is on the plane, we consider it passes through the surface without interactions). Otherwise, we can compute the intersection point P with $t = \frac{(P_1 - \mathbf{o}) \cdot \mathbf{n}}{\mathbf{n} \cdot \mathbf{d}}$. If t lies outside the `min_t` and `max_t` fields of the ray, we also claim that there is no intersection. If t is a valid intersection, we set the `max_t` field of the ray to t, since we only want to consider future intersections happening before this one. Recall that barycentric coordinates $\alpha, \beta, \gamma \in \mathbb{R}$ are the ratio of the signed area of each sub-triangle to the area of the whole triangle. Let $\mathbf{n}_1 = (P_2 - P) \times (P_3 - P)$. We know $\mathbf{n}_1$ is parallel to $\mathbf{n}$ since they are both perpendicular to the plane the triangle $\Delta P_1 P_2 P_3$ is in, so we have $\alpha = \frac{\mathbf{n}_1 \cdot \mathbf{n}}{\mathbf{n} \cdot \mathbf{n}}$. Similarly, if $\mathbf{n}_2 = (P_3 - P) \times (P_1 - P)$, we have $\beta = \frac{\mathbf{n}_2 \cdot \mathbf{n}}{\mathbf{n} \cdot \mathbf{n}}$. Then we have $\gamma = 1 - \alpha - \beta$. With the barycentric coordinates, we can interpolate the normal vector of the intersection point with the weighted normals of three vertices: `isect->n = (alpha * n1 + beta * n2 + gamma * n3).unit()`. We also want to fill the fields `bsdf`, `primitive`, and `t` with the corresponding values of this intersection.

We can also consider the ray-sphere intersection. For a ray $\mathbf{o} + \mathbf{d}t$ and a sphere $(\mathbf{c} - P)^2 = r^2$ for $\mathbf{o}, \mathbf{d}, \mathbf{c}, P \in \mathbb{R}^3$ and $r \in \mathbb{R}$, we can substitute the point P with the ray parametrization, and get the equation $at^2 + bt + c = 0$ where $a = \mathbf{d} \cdot \mathbf{d}$, $b = 2(\mathbf{o} - \mathbf{c}) \cdot \mathbf{d}$, $c = (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2$. For this quadratic equation, we know if the discriminant $b^2 - 4ac < 0$, then there is no real solution, and there is no intersection. Otherwise, we let $t_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$ and $t_2 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$. Both t_1 and t_2 are well defined since the directional vectors of rays are unit vectors. Then similar to the ray-triangle intersection, we compare t_1 and t_2 with `min_t` and `max_t` and update the ray and the intersection. Note that t_1 is guaranteed

to be smaller or equal to t_2, so we do not need to check t_2 if t_1 falls within the range.

Below we have included pictures with normal shading for some simple geometries.

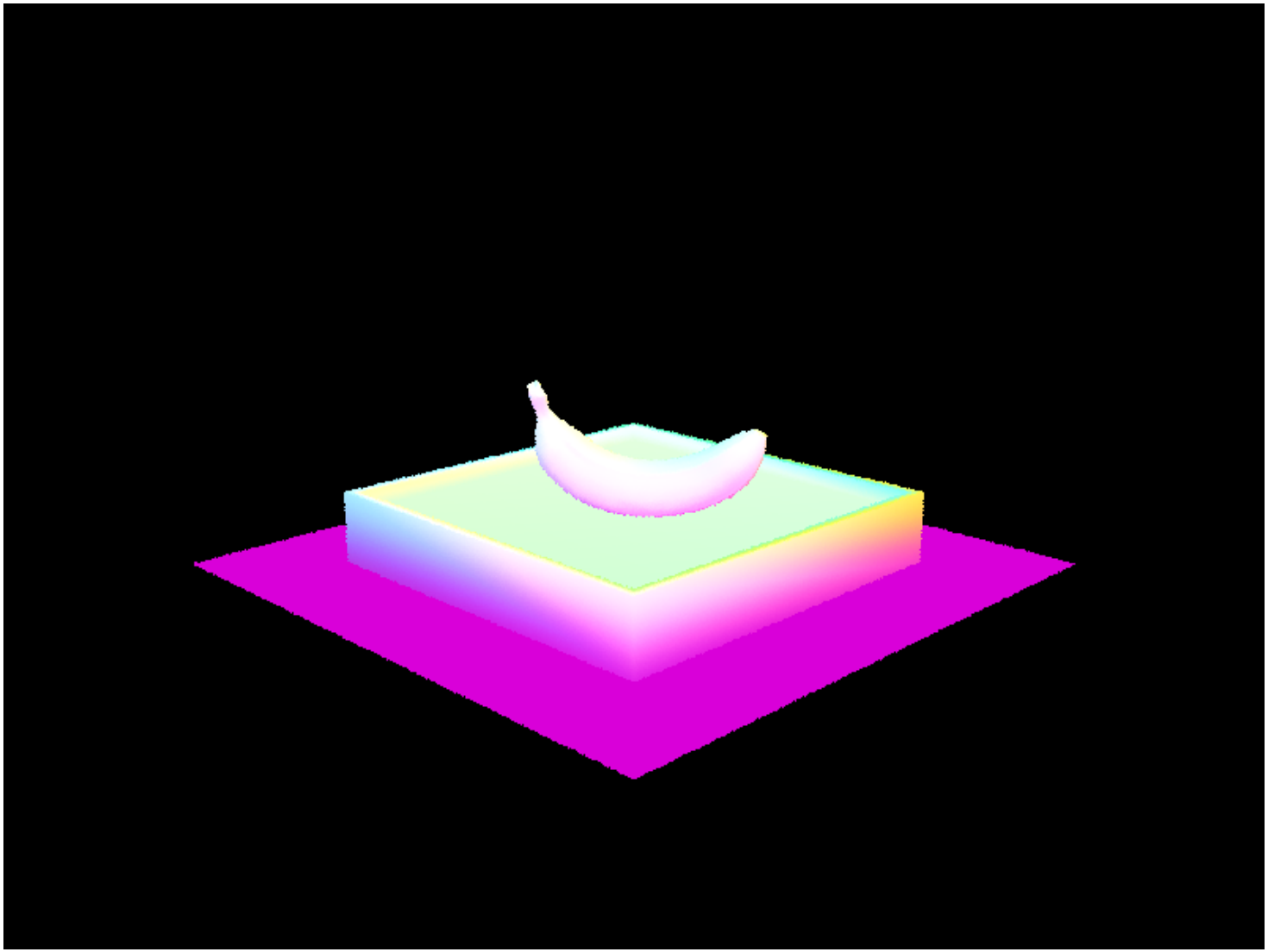

Figure 2: Simple geometry. A banana.

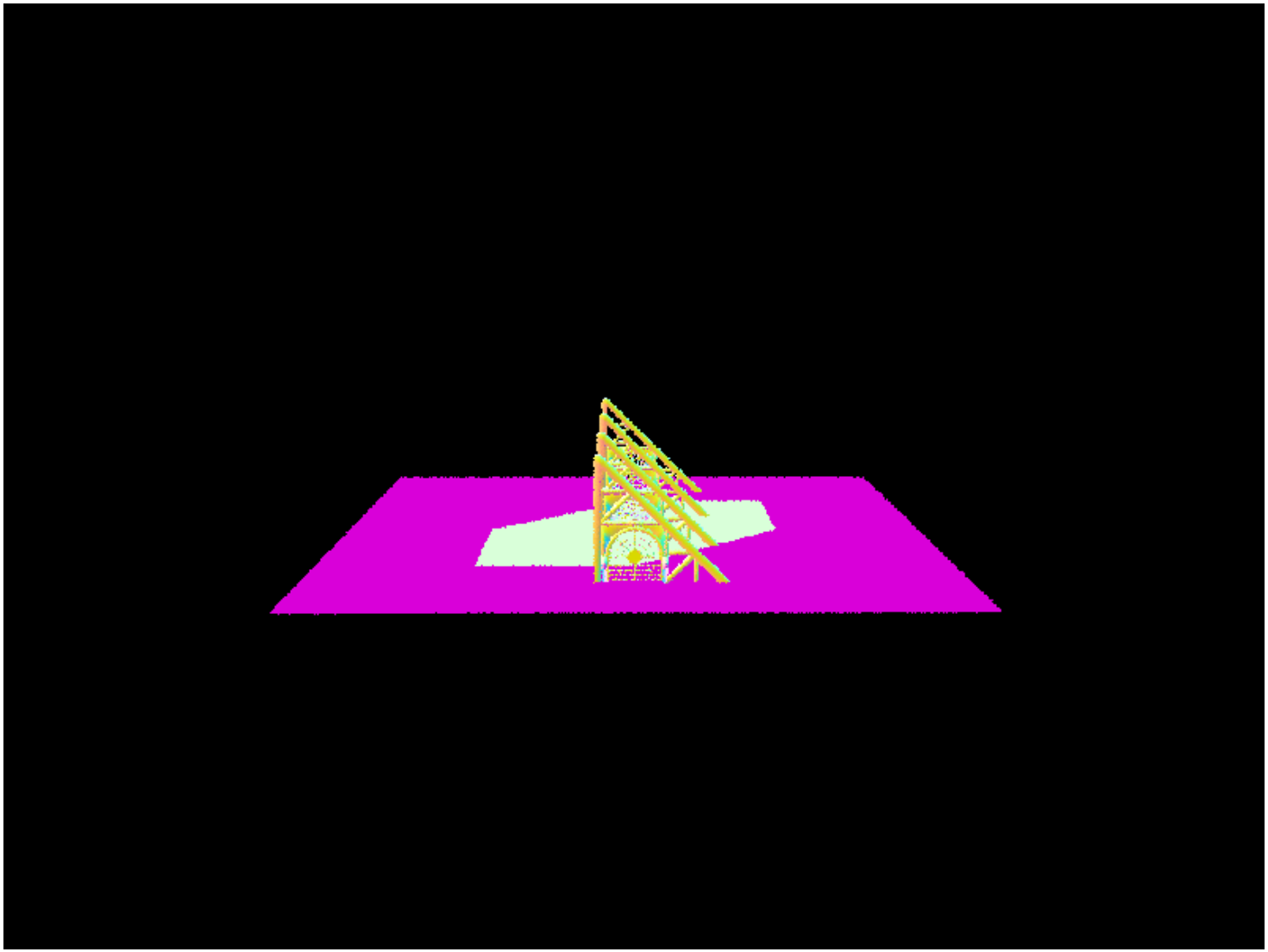

Figure 3: Simple geometry. A building.

Figure 4: Simple geometry. a Utah teapot.

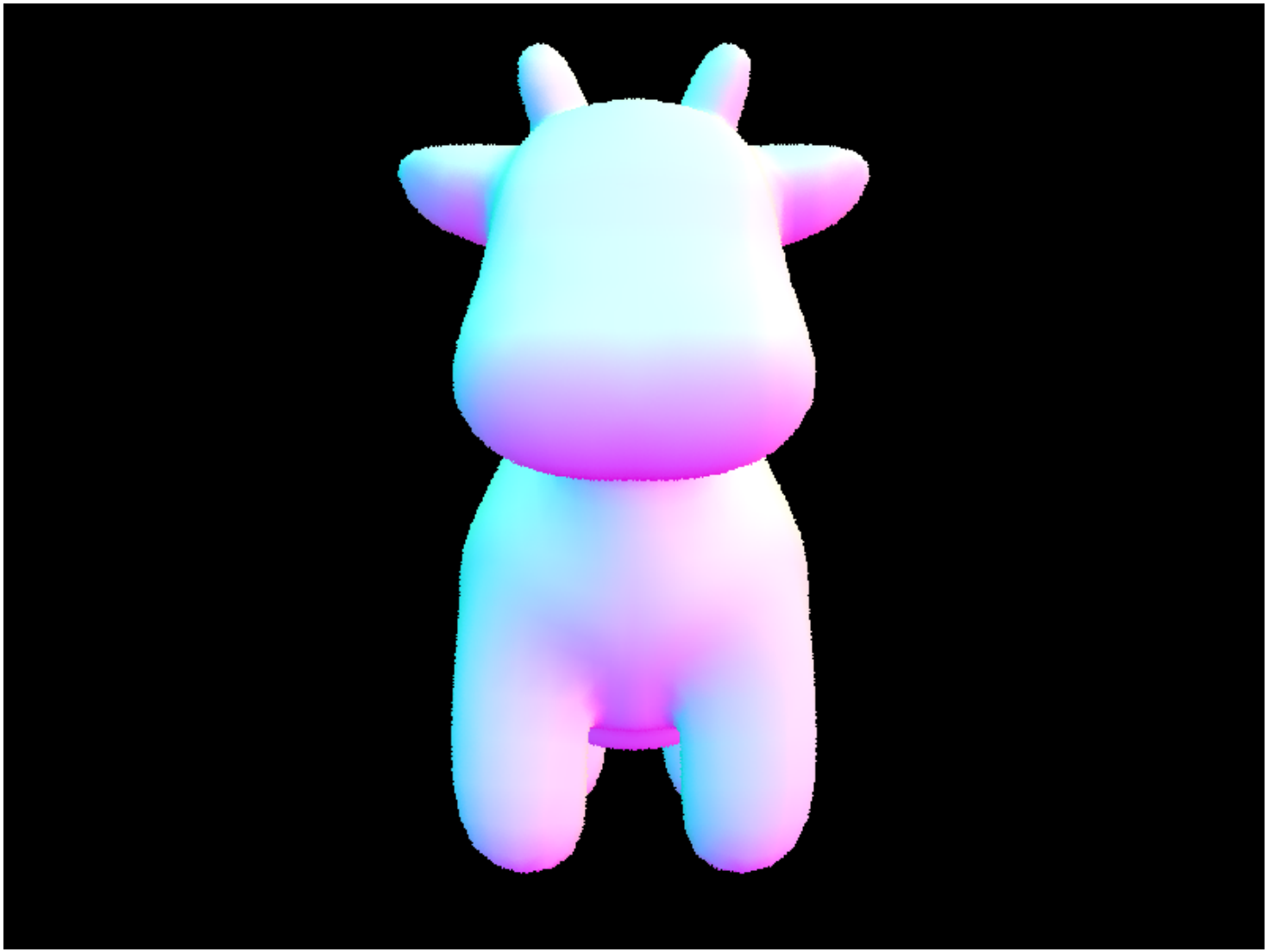

Figure 5: Simple geometry. a cow.

2 Bounding Volume Hierarchy

Bounding Volume Hierarchy (BVH) is a binary tree structure to represent the relations between bounding boxes. The bounding boxes associated with child nodes are contained in the bounding box associated with the parent node. The central idea is repeatedly partitioning the parent bounding boxes into two children bounding boxes and construct a leaf node when the number of primitives is lower than or equal to `max_leaf_size`. All the primitives are stored in a vector and we associate a continuous block of primitives to a node with the iterators `start` and `end` to represent the starting point (inclusive) and the ending point (exclusive). We start a bounding box containing all primitives by iterating through every primitive and getting its bounding box with `BBox bb = (*p)->get_bbox()` and then expand the starting bounding box to contain that bounding box `bbox.expand(bb)`. If the number of primitives inside this bounding box is less than or equal to `max_leaf_size`, then we have finished. We just need to correctly specify the start and the end of the node, set the pointers to the children to null, and return the node.

However, we usually have much more primitives to check intersections than `max_leaf_size`. The heuristic method we use is to partition at the centroid point of the bounding box containing all centroids of the bounding boxes of the primitives associated with this node (`cbbox.expand((*p)->get_bbox().centroid())`) along an axis such that the number of the primitives are most evenly distributed between two partitions. This heuristic will distribute the primitives more evenly than simply partitioning across the axis with the largest extent. It will also run faster than the surface area heuristic. However, the main downside of this algorithm is that we could get very flat or skinny bounding boxes, requiring checks for intersections over a large area. We first set `int axis = -1`, and `int difference = end - start` (the largest possible difference of the number of primitives between two partitions). For each axis, we initialize two vectors `std::vector<Primitive*> left, right` to keep track of primitives whose associated bounding box's centroids are less than the centroid of `cbbox` in that axis. We track the smallest difference and save the corresponding `left` and `right` vectors. If two axes have partitions with the same difference, then we break the tie by choosing the axis with the lower index (x before y before z). If divisions along all three axes result in primitives falling exclusively into one partition, then we make a leaf node for this pathological case. We then copy the primitives of the `left` vector to the vector of primitives from the function argument starting from the `start` position (assigned to the left child node) followed by the primitives of the `right` vector (assigned to the right child node) to preserve the continuity of the primitive blocks and then recursively construct BVH nodes for the left and right children.

To test the intersection between a ray and a bounding box, we can first check whether the ray is parallel to any of the slab structures by checking whether a certain component of the directional vector of the ray has absolute value less than `std::numeric_limits<double>::epsilon()`. If a certain component of the directional vector is very close to 0 and the component of the origin lies

outside the extent of the bounding box, we conclude the ray is parallel to the slab and there is no intersection. In our `BBox::intersect` function, we want to test the intersections between `t0` and `t1` given as arguments. For each intersection of the ray with the slab structure, we record the t of lower intersection to the set S_1 and of the higher to the set S_2, and we choose $t_{min} = \sup S_1$ and $t_{max} = \inf S_2$. If t_{min} is greater than t_{max} or both lie outside the interval $[t_0, t_1]$, then we claim there is no intersection with the bounding box. Otherwise, we update t_0 to be `std::max(t0, t_min)` and t_1 to be `t1 = std::min(t1, t_max)` and conclude there is an intersection with the bounding box.

For any given node on this BVH tree, if a ray does not intersect with the associated bounding box containing all primitives in this node, then we claim that the ray does not intersect with this node. If this node is the root, then we know the ray does not intersect with any of the primitives without checking individual intersections, reducing the amount of computation. If the node is a leaf, then we check the intersections with each primitive inside that node. If there are intersections, we return the closest intersection. If a node is not a leaf, then we recursively check the intersection of the ray with its child nodes. If our heuristic is good at distributing the number of primitives evenly between children nodes, the depth of the BVH tree is logarithmatic, making the tree traversal efficient.

Below we have included pictures of complex geometry with normal shading rendered with BVH acceleration.

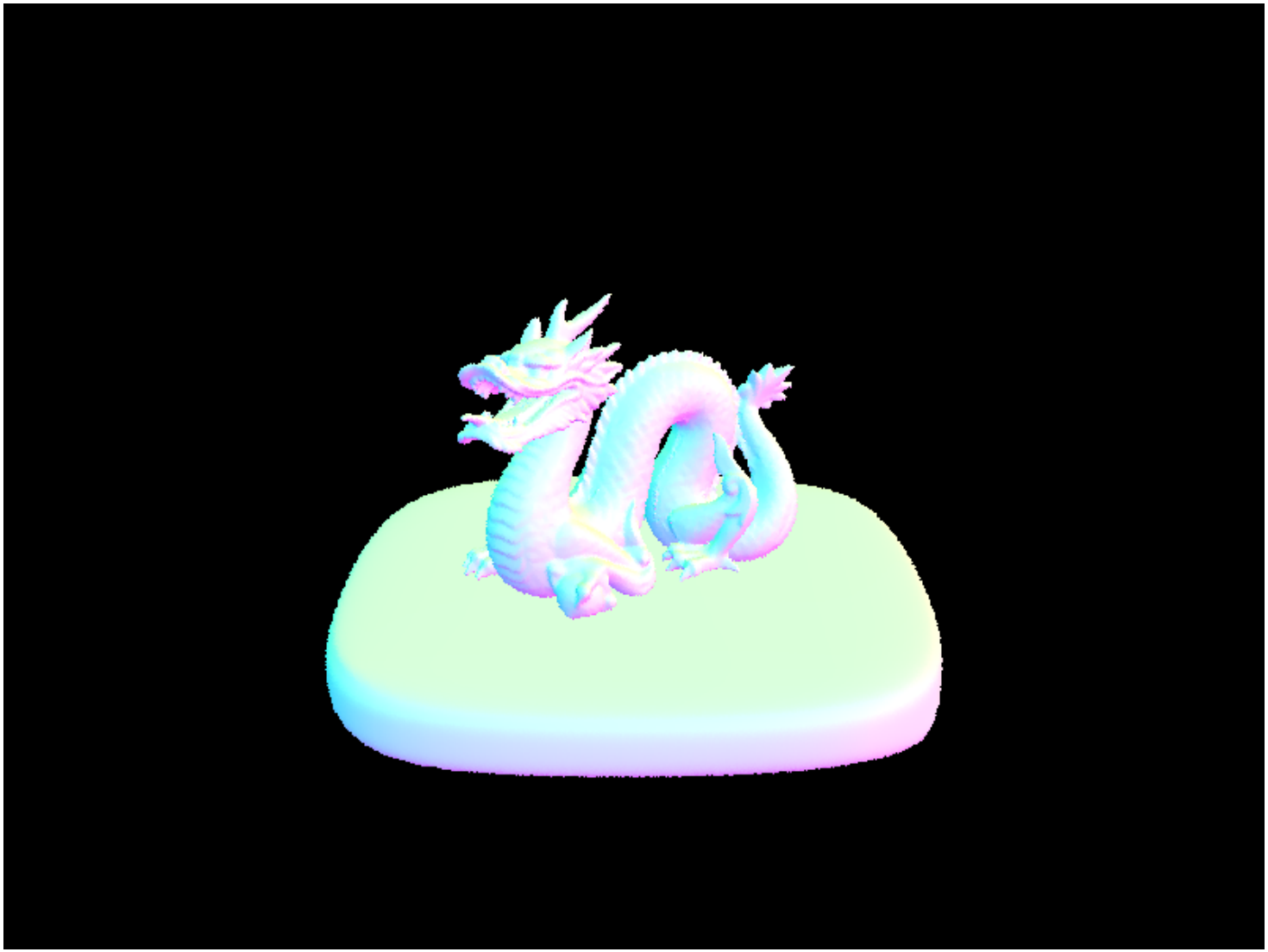

Figure 6: Complex geometry. A dragon. Rendered efficiently with BVH acceleration.

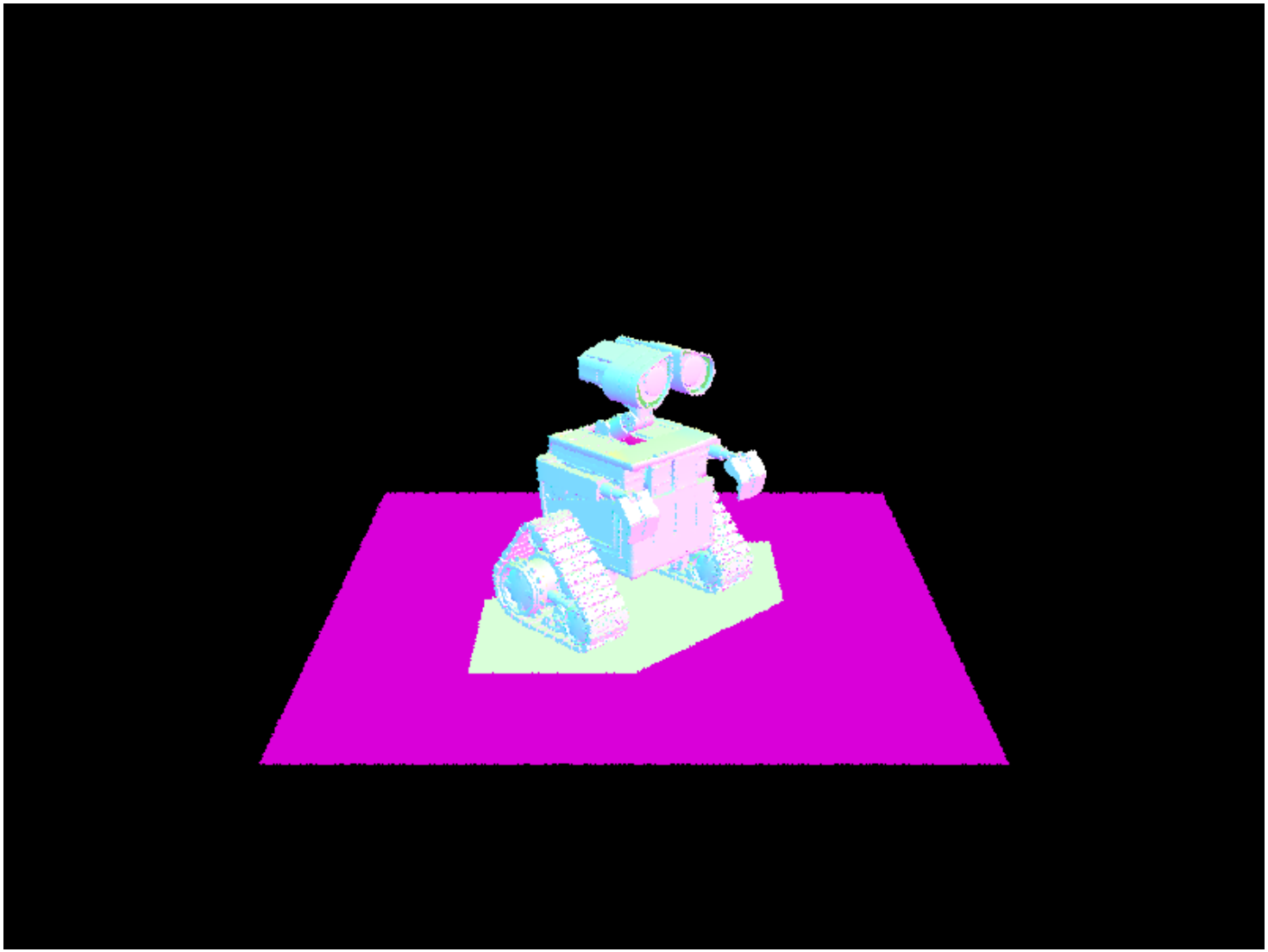

Figure 7: Complex geometry. A robot. Rendered efficiently with BVH acceleration.

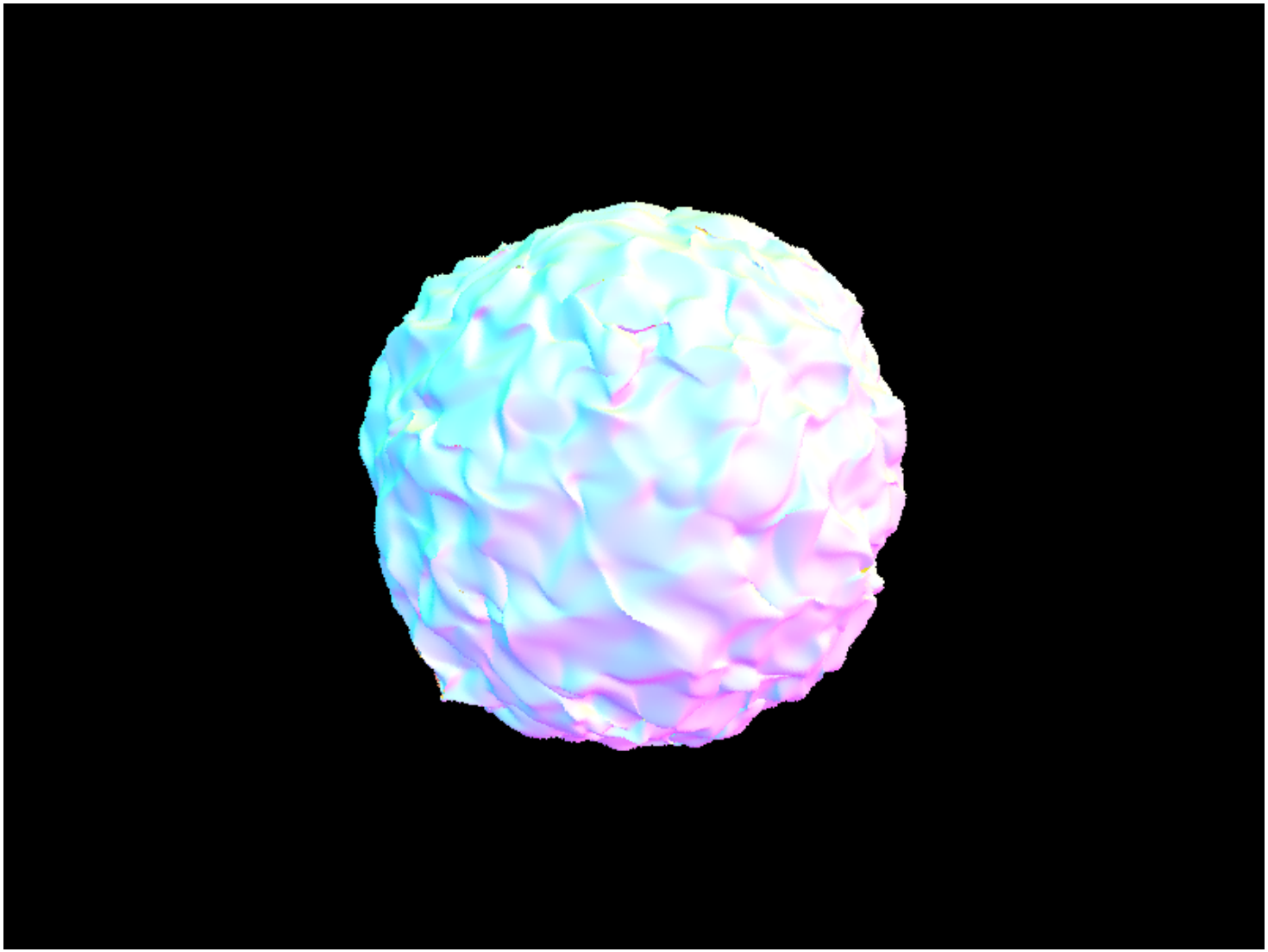

Figure 8: Complex geometry. A blob. Rendered efficiently with BVH acceleration.

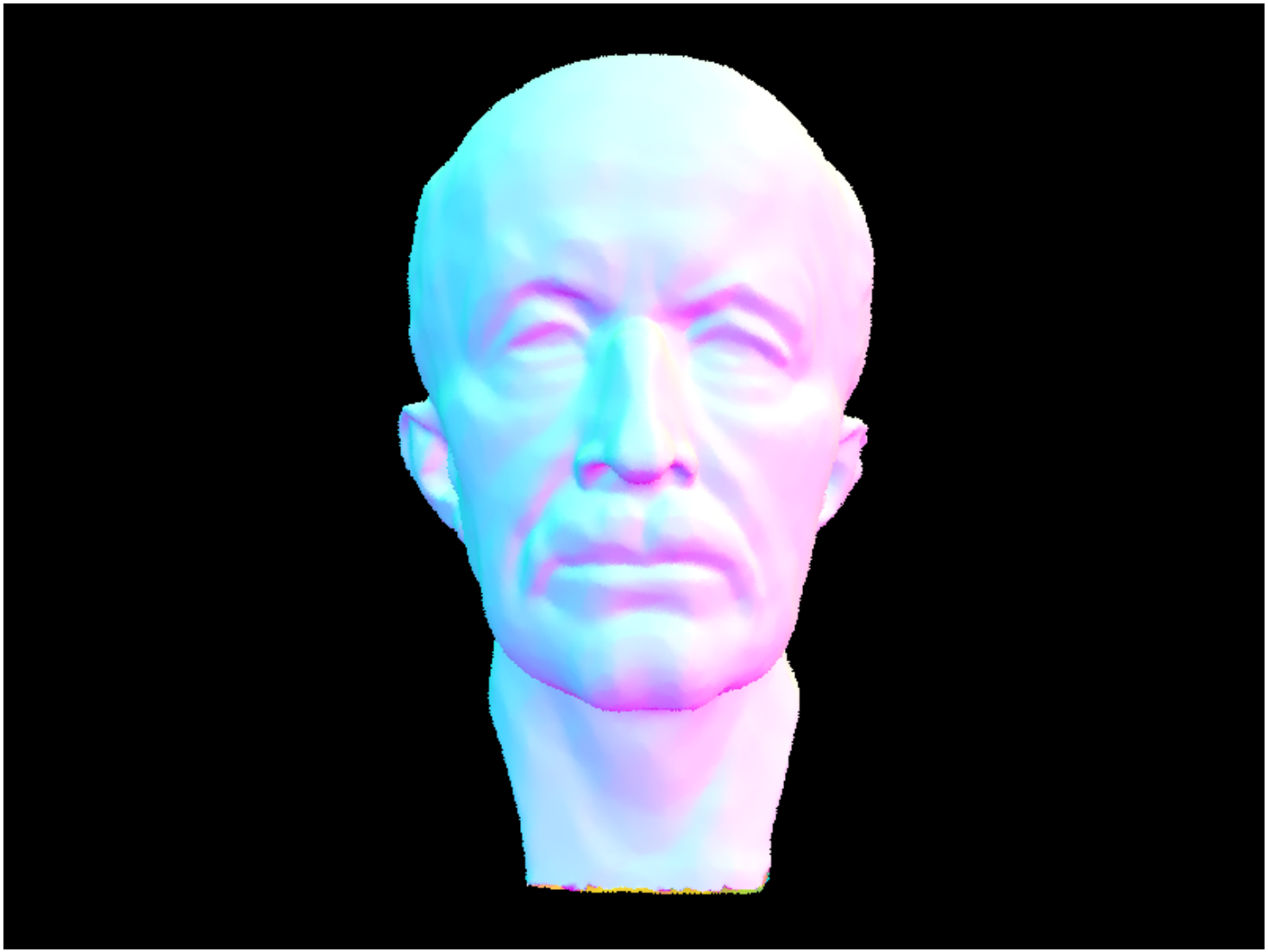

Figure 9: Complex geometry. Max Plank. Rendered efficiently with BVH acceleration.

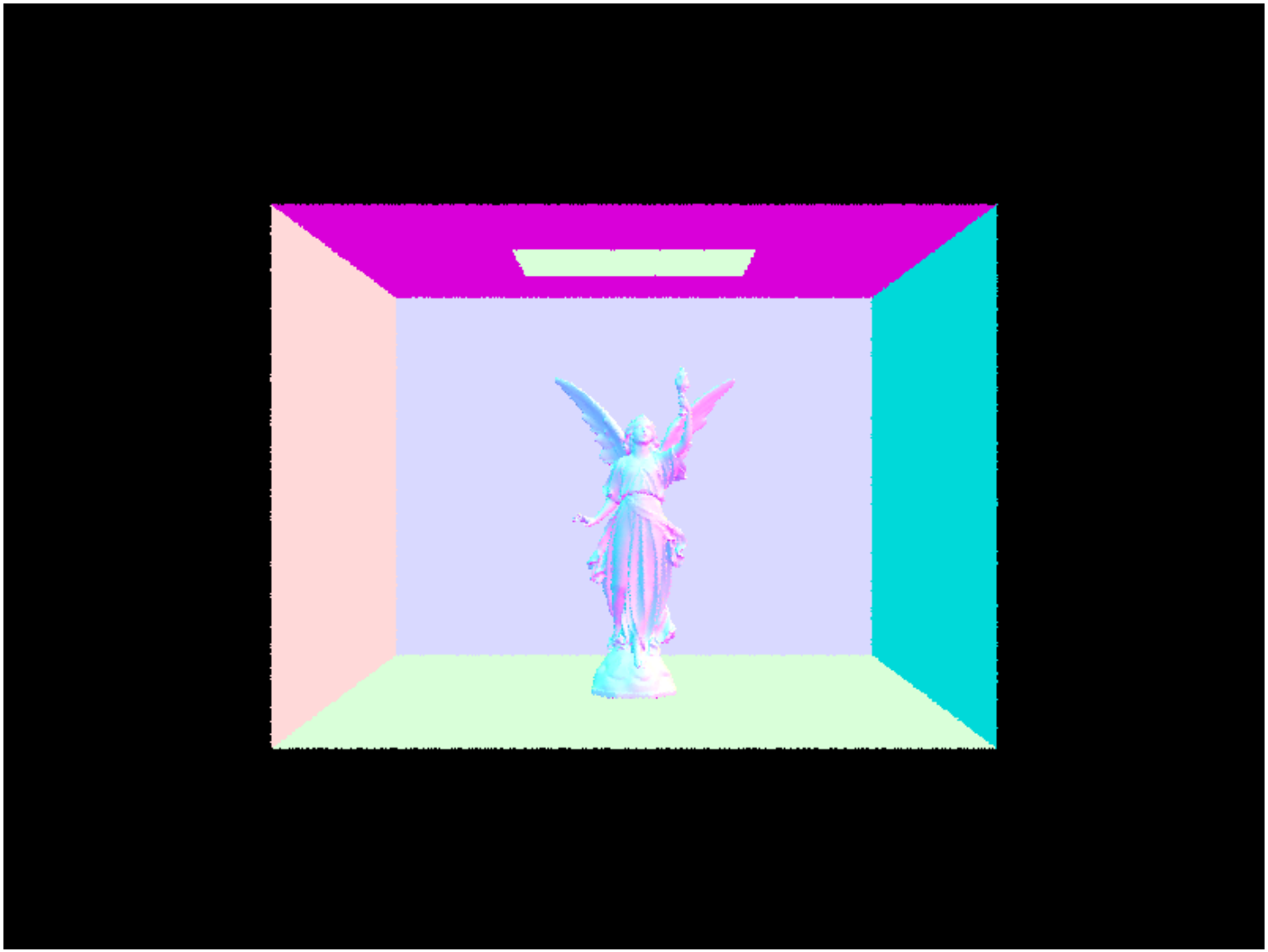

Figure 10: Complex geometry. A statue within a Cornell box. Rendered efficiently with BVH acceleration.

Below we have included the rendering time of some geometries with or without BVH with 1 thread.

Table 1: Rendering Time with and without the BVH Data Structure

	`beetle.dae`	`maxplanck.dae`	`dragon.dae`	`CBluck.dae`
Primitive Numbers	7558	50801	105120	133796
Without BVH	3.0867s	22.9577s	124.5135s	209.5632s
With BVH	0.0977s	0.5341s	0.5015s	0.3846s

We can see that the end results of naive ray-tracing and ray-tracing with the BVH data structure produces identical results. The time to construct BVH tree is also very short, usually well within 1 second, which we can safely ignore in large rendering projects. As the number of primitives increases, the rendering time of naive ray-tracing increases faster than a linear function, making it infeasible for complex scenes with many primitives. In comparison, the rendering time of ray-tracing with the BVH data structure is not only related to the number of primitives, but also the complexity of the geometry that influences how the BVH tree will be constructed. We can see that `CBlucy.dae` has more primitives but takes less time to render than `dragon.dae` since much of the space inside the Cornell box is empty. The time complexity of rendering with BVH is related to the number of primitives, the choice of the heuristic when constructing the BVH tree, and how the primitives are laid out in the scene. In our given examples, using the BVH data structure with only 1 thread drastically reduces the rendering time to below 1 second.

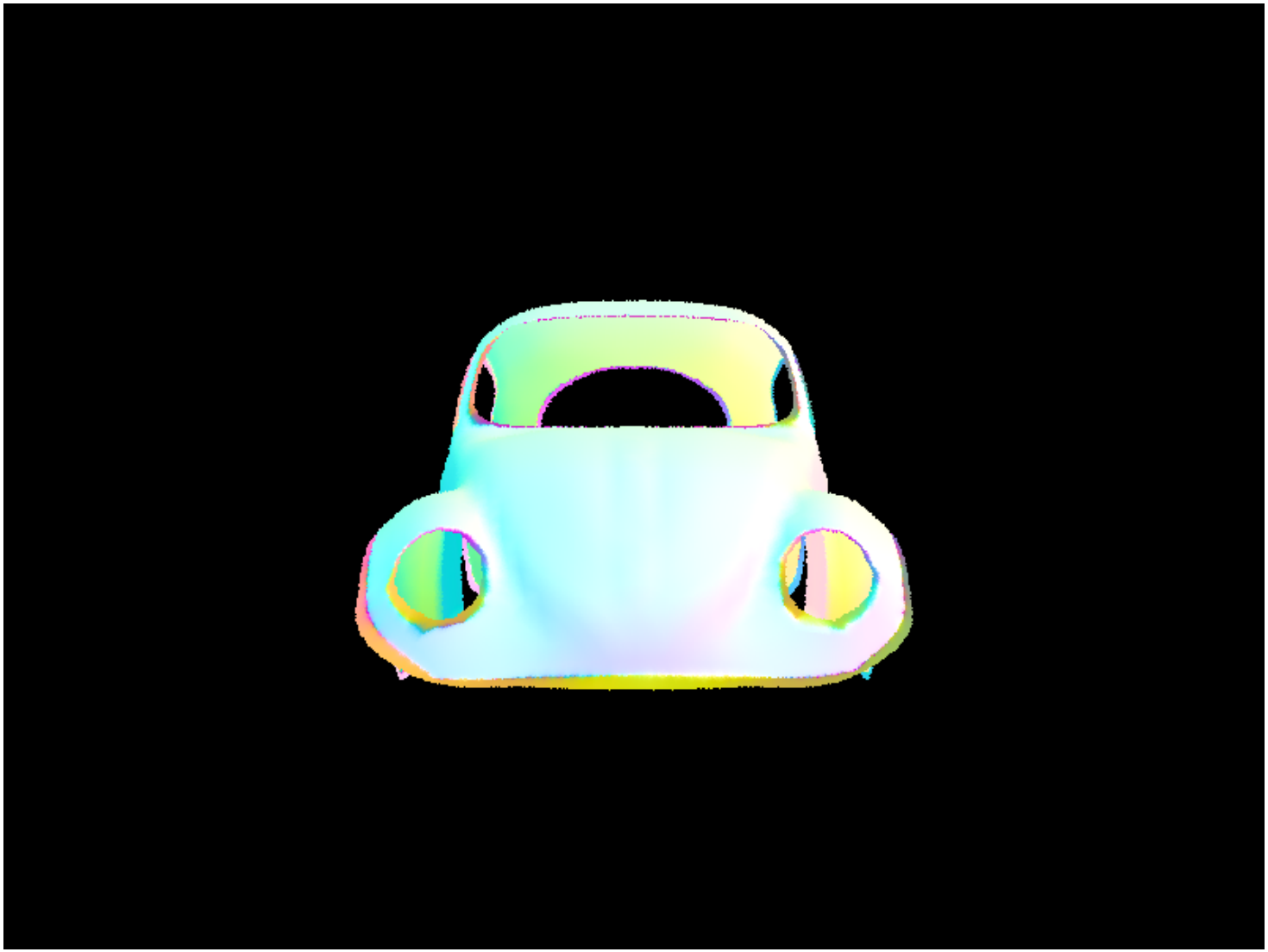

Figure 11: Relatively simple geometry. A Beetle car frame.

```
hongyuli@Hongyus-MacBook-Pro cmake-build-debug % ./pathtracer -r 800 600 -f beetle_s1.png ../dae/meshedit/beetle.dae
[PathTracer] Input scene file: ../dae/meshedit/beetle.dae
[PathTracer] Rendering using 1 threads
[PathTracer] Collecting primitives... Done! (0.0002 sec)
[PathTracer] Building BVH from 7558 primitives... Done! (0.0000 sec)
[PathTracer] Rendering... 100%! (3.0867s)
[PathTracer] BVH traced 480000 rays.
[PathTracer] Average speed 0.1555 million rays per second.
[PathTracer] Averaged 7558.000000 intersection tests per ray.
[PathTracer] Saving to file: beetle_s1.png... Done!
[PathTracer] Job completed.
```

Figure 12: Rendering time of the Beetle car frame without BVH.

```
hongyuli@Hongyus-MacBook-Pro cmake-build-debug % ./pathtracer -r 800 600 -f beetle_bvh.png ../dae/meshedit/beetle.dae
[PathTracer] Input scene file: ../dae/meshedit/beetle.dae
[COLLADA Parser] Loading COLLADA file...
[COLLADA Parser] Loading scene...
[COLLADA Parser]  |- Node: Area_001 (id:Area_001)
[COLLADA Parser]   |- LightInfo: Light (id:Area_001-light) [ type=directional spectrum=(1,1,1) direction=(0,0,-1) ]
[COLLADA Parser]  |- Node: Camera (id:Camera)
[COLLADA Parser]   |- CameraInfo: Camera (id: Camera-camera) [ hfov=39.5978 vfov=22.8952 nclip=0.00999999 fclip=10000 ]
[COLLADA Parser]  |- Node: Mesh_001 (id:Mesh_001)
[COLLADA Parser]   |- PolymeshInfo: Mesh.002 (id:Mesh_002-mesh) [ num_polygons=7558 num_vertices=4063 num_normals=7558 num_texcoords=22674 ]
[COLLADA Parser]   |- MaterialInfo: Default_002 (id:Default_002-material) [ BSDF=0x600001688840 ]
[PathTracer] Rendering using 1 threads
[PathTracer] Collecting primitives... Done! (0.0002 sec)
[PathTracer] Building BVH from 7558 primitives... Done! (0.0038 sec)
[PathTracer] Rendering... 100%! (0.0977s)
[PathTracer] BVH traced 480000 rays.
[PathTracer] Average speed 4.9126 million rays per second.
[PathTracer] Averaged 2.863473 intersection tests per ray.
[PathTracer] Saving to file: beetle_bvh.png... Done!
[PathTracer] Job completed.
```

Figure 13: Rendering time of the Beetle car frame with BVH.

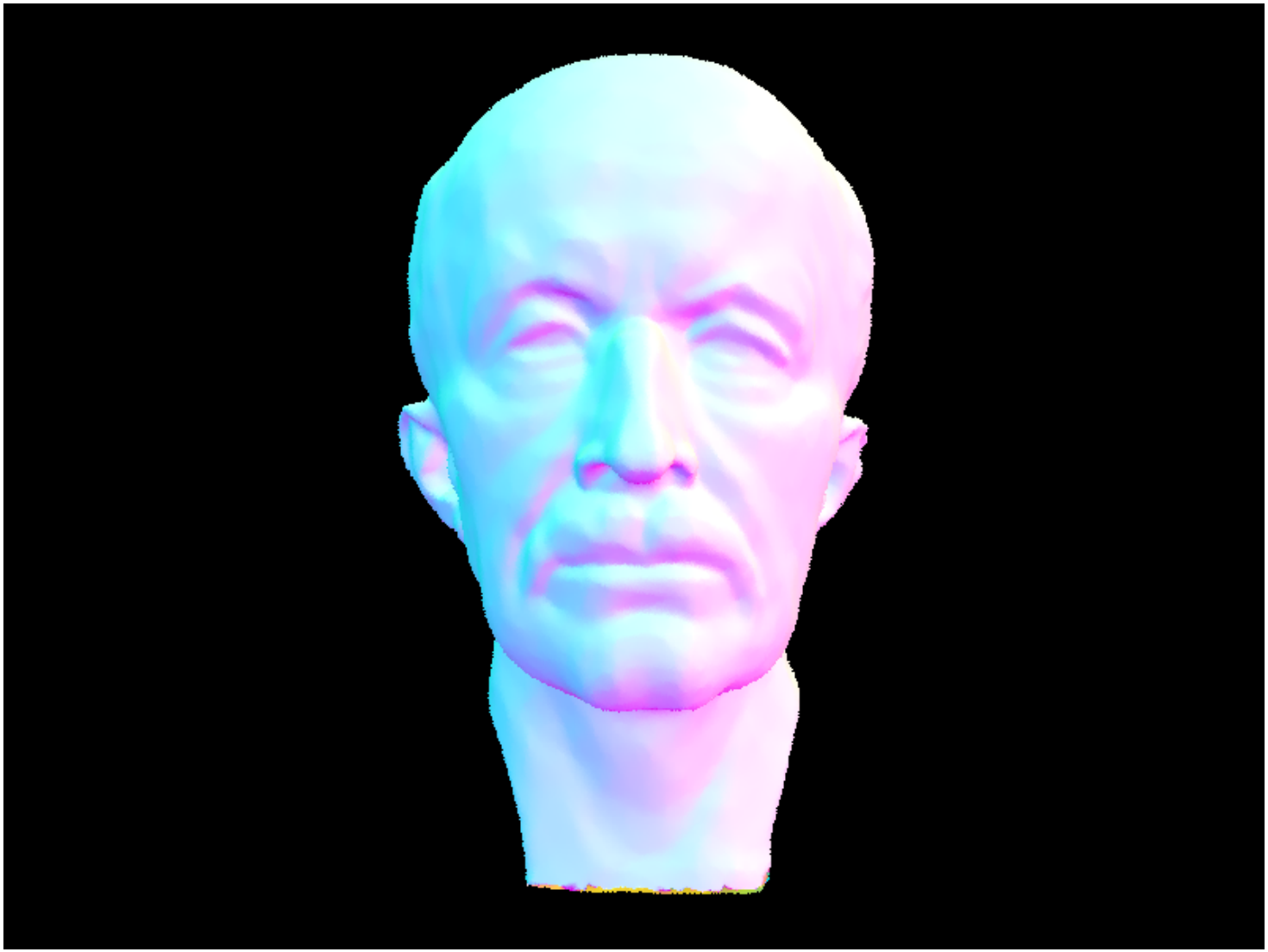

Figure 14: Moderately complex geometry. Max Planck.

```
hongyuli@Hongyus-MacBook-Pro cmake-build-debug % ./pathtracer -r 800 600 -f maxplanck_s1.png ../dae/meshedit/maxplanck.dae
[PathTracer] Input scene file: ../dae/meshedit/maxplanck.dae
[PathTracer] Rendering using 1 threads
[PathTracer] Collecting primitives... Done! (0.0020 sec)
[PathTracer] Building BVH from 50801 primitives... Done! (0.0012 sec)
[PathTracer] Rendering... 100%! (22.9577s)
[PathTracer] BVH traced 480000 rays.
[PathTracer] Average speed 0.0209 million rays per second.
[PathTracer] Averaged 50801.000000 intersection tests per ray.
[PathTracer] Saving to file: maxplanck_s1.png... Done!
[PathTracer] Job completed.
```

Figure 15: Rendering time of Max Planck without BVH.

```
hongyuli@Hongyus-MacBook-Pro cmake-build-debug % ./pathtracer -r 800 600 -f maxplanck_bvh.png ../dae/meshedit/maxplanck.dae
[PathTracer] Input scene file: ../dae/meshedit/maxplanck.dae
[COLLADA Parser] Loading COLLADA file...
[COLLADA Parser] Loading scene...
[COLLADA Parser]  |- Node: Area_001 (id:Area_001)
[COLLADA Parser]   |- LightInfo: Light (id:Area_001-light) [ type=directional spectrum=(1,1,1) direction=(0,0,-1) ]
[COLLADA Parser]  |- Node: Camera (id:Camera)
[COLLADA Parser]   |- CameraInfo: Camera (id: Camera-camera) [ hfov=39.5978 vfov=22.8952 nclip=0.00999999 fclip=10000 ]
[COLLADA Parser]  |- Node: Mesh (id:Mesh)
[COLLADA Parser]   |- PolymeshInfo: Mesh.004 (id:Mesh_004-mesh) [ num_polygons=50801 num_vertices=25445 num_normals=25676 num_texcoords=0 ]
[COLLADA Parser]   |- MaterialInfo: Default_004 (id:Default_004-material) [ BSDF=0x6000030a4040 ]
[PathTracer] Rendering using 1 threads
[PathTracer] Collecting primitives... Done! (0.0018 sec)
[PathTracer] Building BVH from 50801 primitives... Done! (0.0262 sec)
[PathTracer] Rendering... 100%! (0.5341s)
[PathTracer] BVH traced 480000 rays.
[PathTracer] Average speed 0.8987 million rays per second.
[PathTracer] Averaged 29.688950 intersection tests per ray.
[PathTracer] Saving to file: maxplanck_bvh.png... Done!
[PathTracer] Job completed.
```

Figure 16: Rendering time of the Max Planck with BVH.

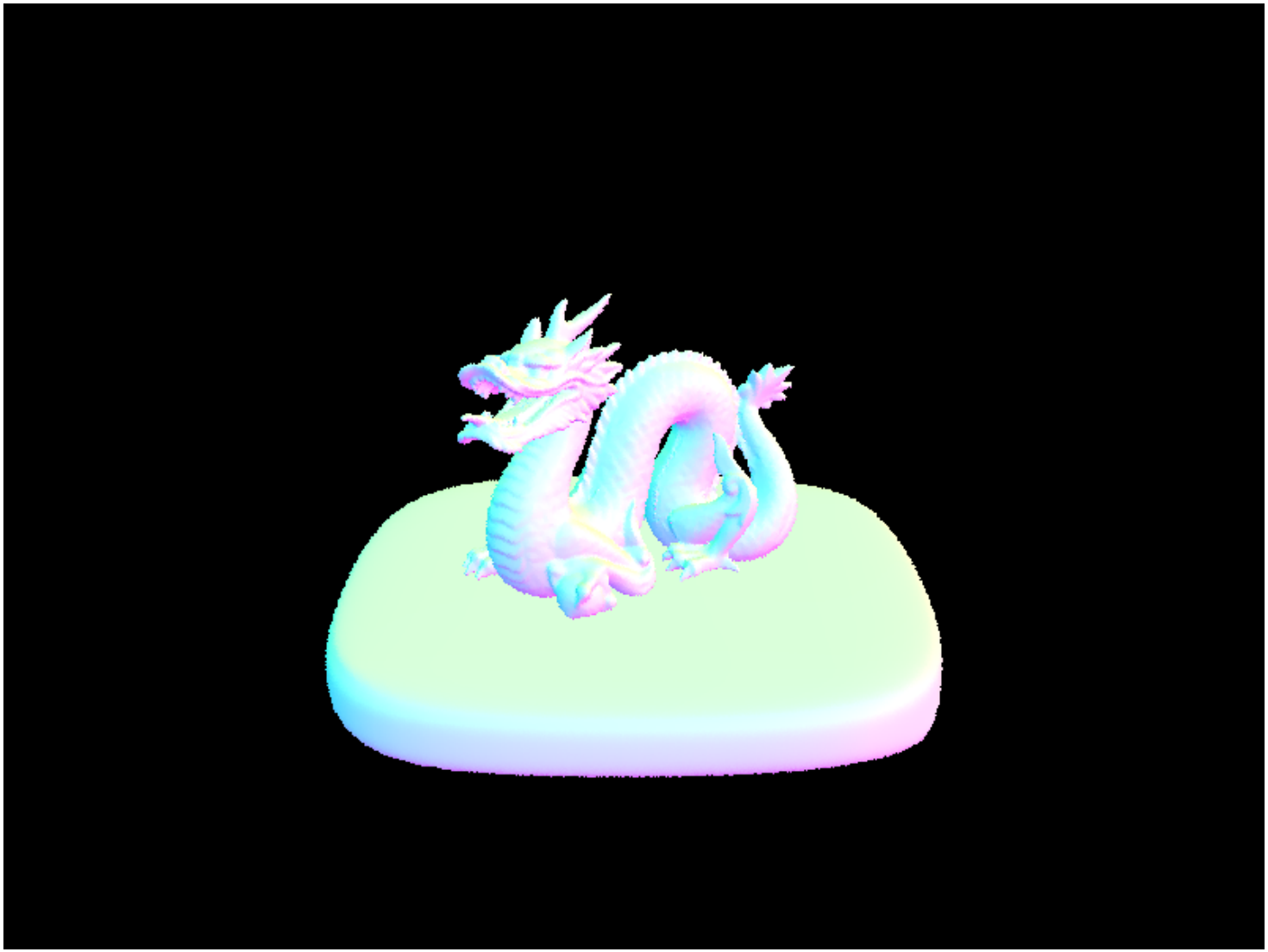

Figure 17: Moderate to complex geometry. A dragon.

```
hongyuli@Hongyus-MacBook-Pro cmake-build-debug % ./pathtracer -r 800 600 -f dragon_s1.png ../dae/sky/dragon.dae
[PathTracer] Input scene file: ../dae/sky/dragon.dae
[PathTracer] Rendering using 1 threads
[PathTracer] Collecting primitives... Done! (0.0072 sec)
[PathTracer] Building BVH from 105120 primitives... Done! (0.0028 sec)
[PathTracer] Rendering... 100%! (124.5135s)
[PathTracer] BVH traced 480000 rays.
[PathTracer] Average speed 0.0039 million rays per second.
[PathTracer] Averaged 105120.000000 intersection tests per ray.
[PathTracer] Saving to file: dragon_s1.png... Done!
[PathTracer] Job completed.
```

Figure 18: Rendering time of the dragon without BVH.

```
hongyuli@Hongyus-MacBook-Pro cmake-build-debug % ./pathtracer -r 800 600 -f dragon_bvh.png ../dae/sky/dragon.dae
[PathTracer] Input scene file: ../dae/sky/dragon.dae
[COLLADA Parser] Loading COLLADA file...
[COLLADA Parser] Loading scene...
[COLLADA Parser]  |- Node: Hemi (id:Hemi)
[COLLADA Parser]   |- LightInfo: Hemi (id:Hemi-light) [ type=ambient spectrum=(1,1,1) ]
[COLLADA Parser]  |- Node: Area_001 (id:Area_001)
[COLLADA Parser]   |- LightInfo: Light (id:Area_001-light) [ type=directional spectrum=(1,1,1) direction=(0,0,-1) ]
[COLLADA Parser]  |- Node: Camera (id:Camera)
[COLLADA Parser]   |- CameraInfo: Camera.001 (id: Camera-camera) [ hfov=49.1343 vfov=28.8415 nclip=0.1 fclip=100 ]
[COLLADA Parser]  |- Node: dragon (id:dragon)
[COLLADA Parser]   |- PolymeshInfo: dragon (id:dragon-mesh) [ num_polygons=100000 num_vertices=50000 num_normals=100000 num_texcoords=0 ]
[COLLADA Parser]   |- MaterialInfo: dragon (id:dragon-material) [ BSDF=0x600003718040 ]
[COLLADA Parser]  |- Node: Cube (id:Cube)
[COLLADA Parser]   |- PolymeshInfo: Cube (id:Cube-mesh) [ num_polygons=5120 num_vertices=2562 num_normals=5120 num_texcoords=0 ]
[COLLADA Parser]   |- MaterialInfo: pedestal (id:pedestal-material) [ BSDF=0x6000037180c0 ]
[PathTracer] Rendering using 1 threads
[PathTracer] Collecting primitives... Done! (0.0133 sec)
[PathTracer] Building BVH from 105120 primitives... Done! (0.0720 sec)
[PathTracer] Rendering... 100%! (0.5015s)
[PathTracer] BVH traced 480000 rays.
[PathTracer] Average speed 0.9572 million rays per second.
[PathTracer] Averaged 20.845508 intersection tests per ray.
[PathTracer] Saving to file: dragon_bvh.png... Done!
[PathTracer] Job completed.
```

Figure 19: Rendering time of the the dragon with BVH.

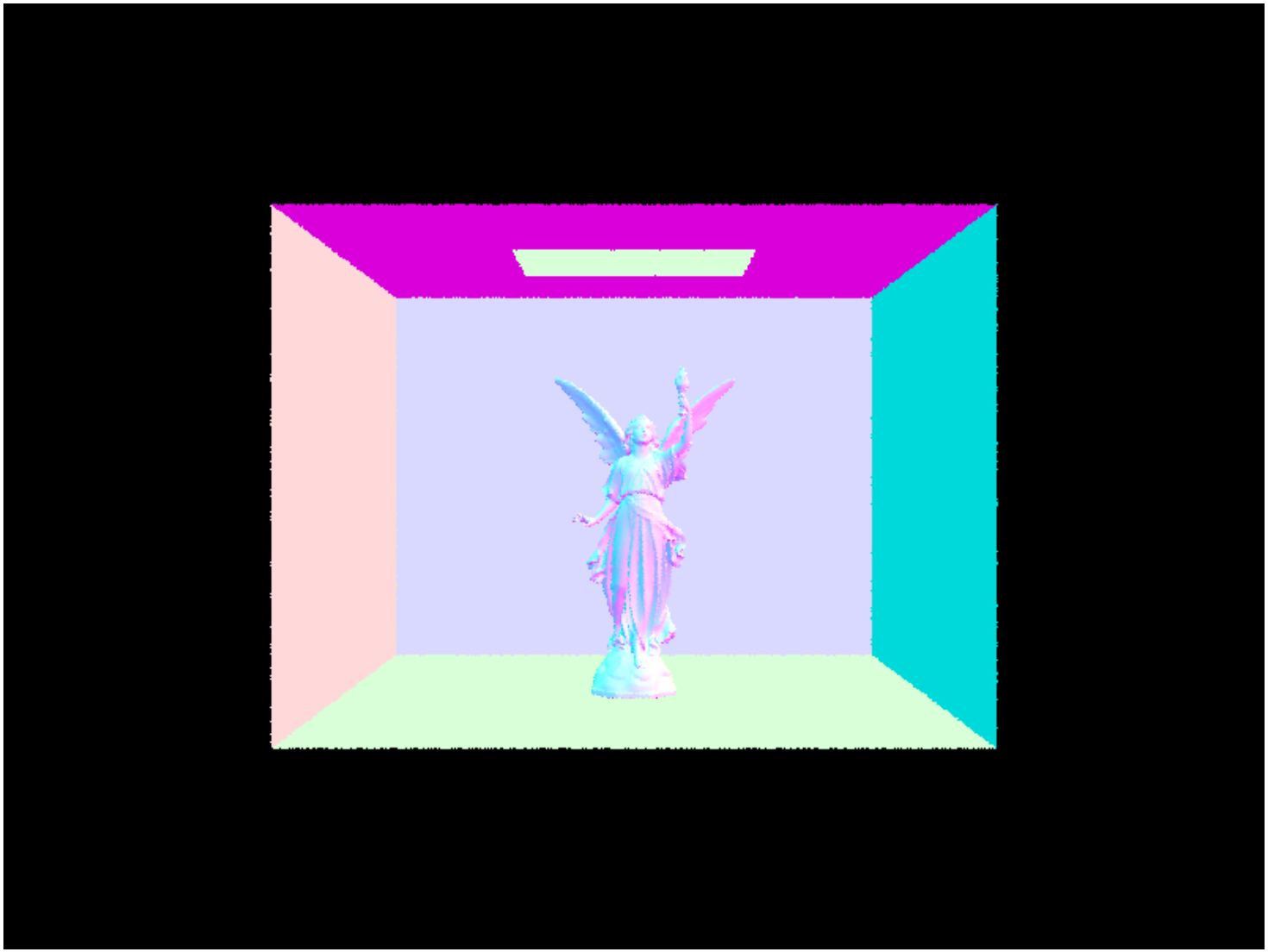

Figure 20: Complex geometry. A statue in a Cornell box.

```
hongyuli@Hongyus-MacBook-Pro cmake-build-debug % ./pathtracer -r 800 600 -f CBlucy_s1.png ../dae/sky/CBlucy.dae
[PathTracer] Input scene file: ../dae/sky/CBlucy.dae
[PathTracer] Rendering using 1 threads
[PathTracer] Collecting primitives... Done! (0.0105 sec)
[PathTracer] Building BVH from 133796 primitives... Done! (0.0023 sec)
[PathTracer] Rendering... 100%! (209.5632s)
[PathTracer] BVH traced 480000 rays.
[PathTracer] Average speed 0.0023 million rays per second.
[PathTracer] Averaged 133796.000000 intersection tests per ray.
[PathTracer] Saving to file: CBlucy_s1.png... Done!
[PathTracer] Job completed.
```

Figure 21: Rendering time of the statue without BVH.

```
hongyuli@Hongyus-MacBook-Pro cmake-build-debug % ./pathtracer -r 800 600 -f CBlucy_bvh.png ../dae/sky/CBlucy.dae
[PathTracer] Input scene file: ../dae/sky/CBlucy.dae
[COLLADA Parser] Loading COLLADA file...
[COLLADA Parser] Loading scene...
[COLLADA Parser]  |- Node: Lucy (id:Lucy)
[COLLADA Parser]   |- PolymeshInfo: Lucy (id:Lucy-mesh) [ num_polygons=133784 num_vertices=66892 num_normals=67142 num_texcoords=401352 ]
[COLLADA Parser]   |- MaterialInfo: Lucy (id:Lucy-material) [ BSDF=0x60000312c000 ]
[COLLADA Parser]  |- Node: Camera (id:Camera)
[COLLADA Parser]   |- CameraInfo: Camera (id: Camera-camera) [ hfov=49.1343 vfov=28.8415 nclip=0.1 fclip=100 ]
[COLLADA Parser]  |- Node: Area (id:Area)
[COLLADA Parser]   |- LightInfo: Light (id:Area-light) [ type=area spectrum=(10,10,10) direction=(0,0,-1) ]
[COLLADA Parser]  |- Node: ceiling (id:ceiling)
[COLLADA Parser]   |- PolymeshInfo: ceiling (id:ceiling-mesh) [ num_polygons=2 num_vertices=4 num_normals=2 num_texcoords=6 ]
[COLLADA Parser]   |- MaterialInfo: ceiling (id:ceiling-material) [ BSDF=0x600000024080 ]
[COLLADA Parser]  |- Node: light (id:light)
[COLLADA Parser]   |- PolymeshInfo: light (id:light-mesh) [ num_polygons=2 num_vertices=4 num_normals=2 num_texcoords=6 ]
[COLLADA Parser]   |- MaterialInfo: light (id:light-material) [ BSDF=0x6000000240c0 ]
[COLLADA Parser]  |- Node: floor (id:floor)
[COLLADA Parser]   |- PolymeshInfo: floor (id:floor-mesh) [ num_polygons=2 num_vertices=4 num_normals=2 num_texcoords=6 ]
[COLLADA Parser]   |- MaterialInfo: floor (id:floor-material) [ BSDF=0x600000024180 ]
[COLLADA Parser]  |- Node: leftWall (id:leftWall)
[COLLADA Parser]   |- PolymeshInfo: leftWall (id:leftWall-mesh) [ num_polygons=2 num_vertices=4 num_normals=2 num_texcoords=6 ]
[COLLADA Parser]   |- MaterialInfo: leftWall (id:leftWall-material) [ BSDF=0x600000024200 ]
[COLLADA Parser]  |- Node: rightWall (id:rightWall)
[COLLADA Parser]   |- PolymeshInfo: rightWall (id:rightWall-mesh) [ num_polygons=2 num_vertices=4 num_normals=2 num_texcoords=6 ]
[COLLADA Parser]   |- MaterialInfo: rightWall (id:rightWall-material) [ BSDF=0x600000024280 ]
[COLLADA Parser]  |- Node: backWall (id:backWall)
[COLLADA Parser]   |- PolymeshInfo: backWall (id:backWall-mesh) [ num_polygons=2 num_vertices=4 num_normals=2 num_texcoords=6 ]
[COLLADA Parser]   |- MaterialInfo: backWall (id:backWall-material) [ BSDF=0x600000024300 ]
[PathTracer] Rendering using 1 threads
[PathTracer] Collecting primitives... Done! (0.0202 sec)
[PathTracer] Building BVH from 133796 primitives... Done! (0.0813 sec)
[PathTracer] Rendering... 100%! (0.3846s)
[PathTracer] BVH traced 480000 rays.
[PathTracer] Average speed 1.2482 million rays per second.
[PathTracer] Averaged 11.720042 intersection tests per ray.
[PathTracer] Saving to file: CBlucy_bvh.png... Done!
[PathTracer] Job completed.
```

Figure 22: Rendering time of the the statue with BVH.

3 Direct Illumination

For the scope of this assignment, we are working with surfaces with the diffuse BSDF, namely, material that scatters light equally in all directions. Let $\mathbf{p} \in \mathbb{R}^3$ be the hit point on the surface, ω_i be the incoming direction and ω_o be the outgoing direction, the BSDF is given as $f(\mathbf{p}, \omega_i, \omega_o) = \frac{\rho}{\pi}$ where ρ is the reflectance of the material.

When we consider the zero-bounce illumination, the sampling method is irrelevant, since what a viewer sees is determined by the emission of the surface, namely, whether the surface is a light source. Therefore, the zero bounce radiance is always `isect.bsdf->get_emission()`, both in the context of direct illumination and global illumination regardless of the sampling method.

For a given point on the surface, we want to estimate the irradiance on that point. Because the light paths are reversible, in our computation, we treat the light rays as if they were emitted from that point. However, doing formal symbolic integration is computationally infeasible, we use the numerical method Monte Carlo Integration to estimate the integral of the irradiance. Monte Carlo Integration requires us to sample the value of the integrand, divide it over the probability value at that point, and take the average over multiple samples. Since the diffuse BSDF is constant regardless of the incoming and outgoing angles, the most immediate idea is sampling the uniform hemisphere above the surface. If we randomly choose a point on a uniform unit hemisphere, the probability distribution function is $p = \frac{1}{2\pi}$ since the surface area is 2π. In our C++ program, we can get such a random sample of the incoming angle ω_i represented by its Cartesian coordinates in the local coordinate system with `hemisphereSampler->get_sample()`. Note that the inner product of ω_i and the normal vector $(0, 0, 1)$ (i.e. the z-component of ω_i) is the cosine term in the Lambert's Cosine Law. We can get the position w_i in the world coordinate system of ω_i by left multiplying the `o2w` matrix. In this way, we can set the origin of the shadow ray to be the hit point on the surface and the directional vector to be `w_i`. We also set the `min_t` field of the ray to be `EPS_F` to avoid the self-intersection with the surface. Then we can cast this shadow ray and use our BVH data structure to find the closest intersection. Since we are considering the direct bounce, in the rendering equation, we set L_i to be the emission of new intersection (zero if the new intersection is not a light source), so the Monte Carlo estimation term will be `isect.bsdf->f(w_out, omega_i)* Li * omega_i.z * 2.0 * M_PI`. For each light source, we want to take `ns_area_light` samples, so the total number of samples we will take is `scene->lights.size() * ns_area_light`. Applying the Monte Carlo Integration, and we can get an estimation of the radiance on the hit point given by the direct bounce from the light source reflected to the viewer.

However, when we actually apply this uniform hemisphere sampling strategy, we will see that the images generated are quite noisy. The reason is that we are sampling the upper hemisphere with uniform probability regardless of whether there is likely to be a light source or not, wasting much computation resources on regions with no light source. If we already have information about the light

sources, we can use importance sampling to avoid sampling wide regions of no light.

In our C++ program, we distinguish between two different light sources. If a light source is delta light, we only take one sample, since any number of samples will return identical results. If a light source is not delta light, we take `ns_area_light` samples to capture its radiance in relation to the spatial information. We can call the function `sample_L(hit_p, &wi, &distToLight, &pdf)` to get information of the incoming angle `wi` in the world coordinate system, the distance to light source, the probability (used for Monte Carlo Integration), and the incoming radiance `Li`. Similar to the uniform hemisphere sampling, we can construct a shadow ray from the hit point with the directional vector `wi`. We set the `min_t` field to be `EPS_F` to avoid self-intersection and `max_t` to be `distToLight` since the directional vector is a unit vector. In the context of direct lighting, we are only interested in the cases where the shadow ray is not intercepted by other primitives in the range of `min_t` and `max_t`. Without BVH implementation, we can use `has_intersection` method to check the intersection more efficiently. If the light is unblocked, we can use the surface BSDF to get the Monte Carlo estimation term `isect.bsdf->f(w_out, omega_i) * Li * omega_i.z / pdf`. With the extra information about the light source, we avoid generating the samples where there is guaranteed to be no light in the Monte Carlo Integration, making our result much less noisy.

Below we have included pictures of a bunny inside a Cornell box and two spheres inside a Cornell box with the uniform hemisphere sampling and importance sampling.

As we can see, the importance sampling generates a much cleaner image with significantly less noise compared to uniform hemisphere sampling. The edges of the light source on the ceiling are a bit blurry with the uniform hemisphere sampling but appear sharp with importance sampling. Even on the walls where the color is close to uniform, we can still observe considerable noise with uniform hemisphere sampling. As we have discussed earlier, one major reason for the noisy output for the uniform hemisphere sampling is that it samples with uniform probability the whole region above the surface, regardless whether there is light source or not. Therefore, a lot of computational resources are spent on regions with no light sources, while the region close to light sources receive insufficent sampling. Another cause of the high noise is the high variance of the terms used in the numerical Monte Carlo Integration estimation. If the area of the light source is much smaller than the scene, then most of the samples will return no light while only a few numbers of the samples will contain information about the light source. Since the number of valid samples of the light source is small, the statistical variance is large, and this high variance manifests as the noisy pixels in the rendering output.

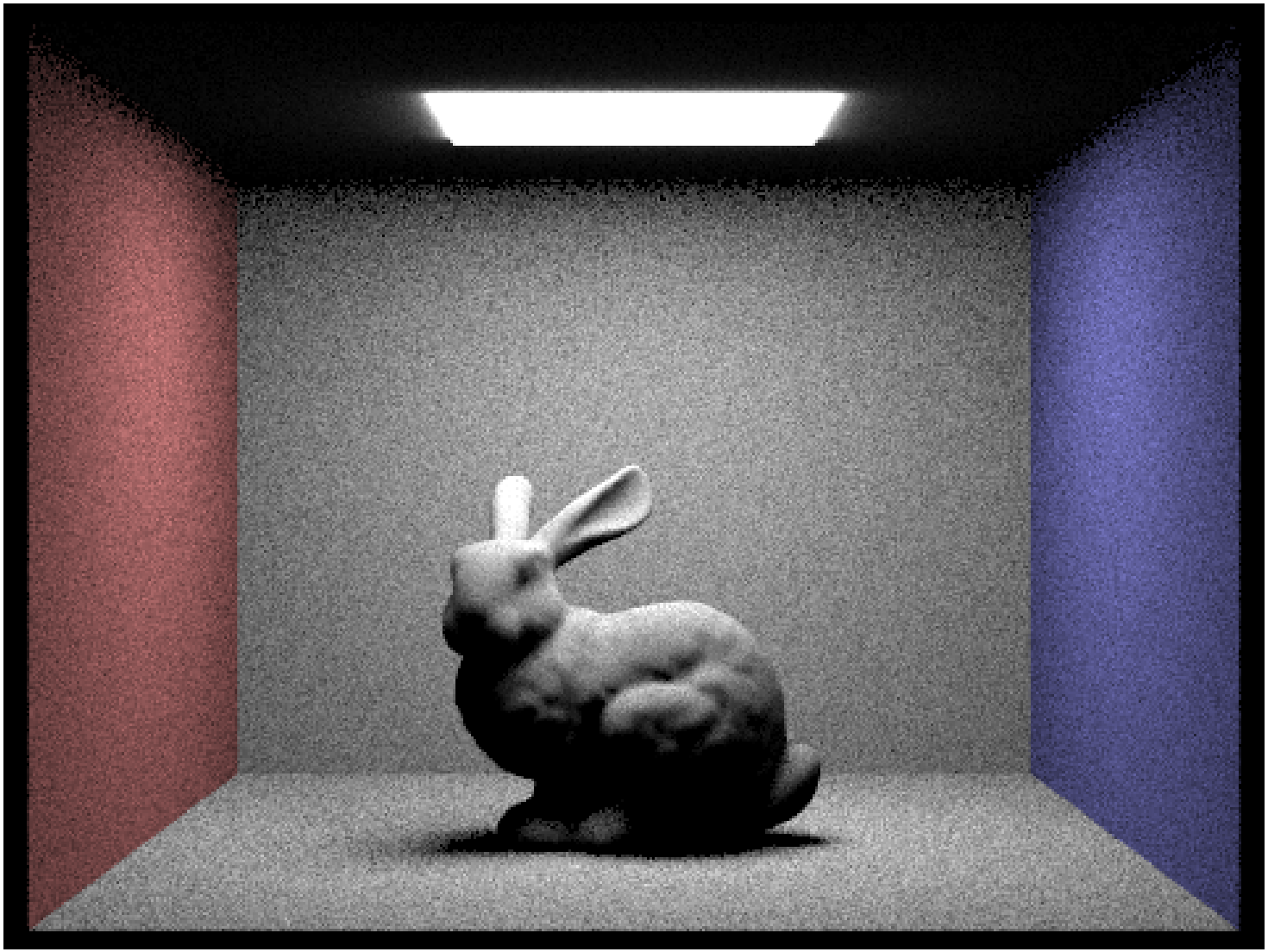

Figure 23: 1024 samples per pixel, 1 samples per area light, direct lighting with uniform hemisphere sampling.

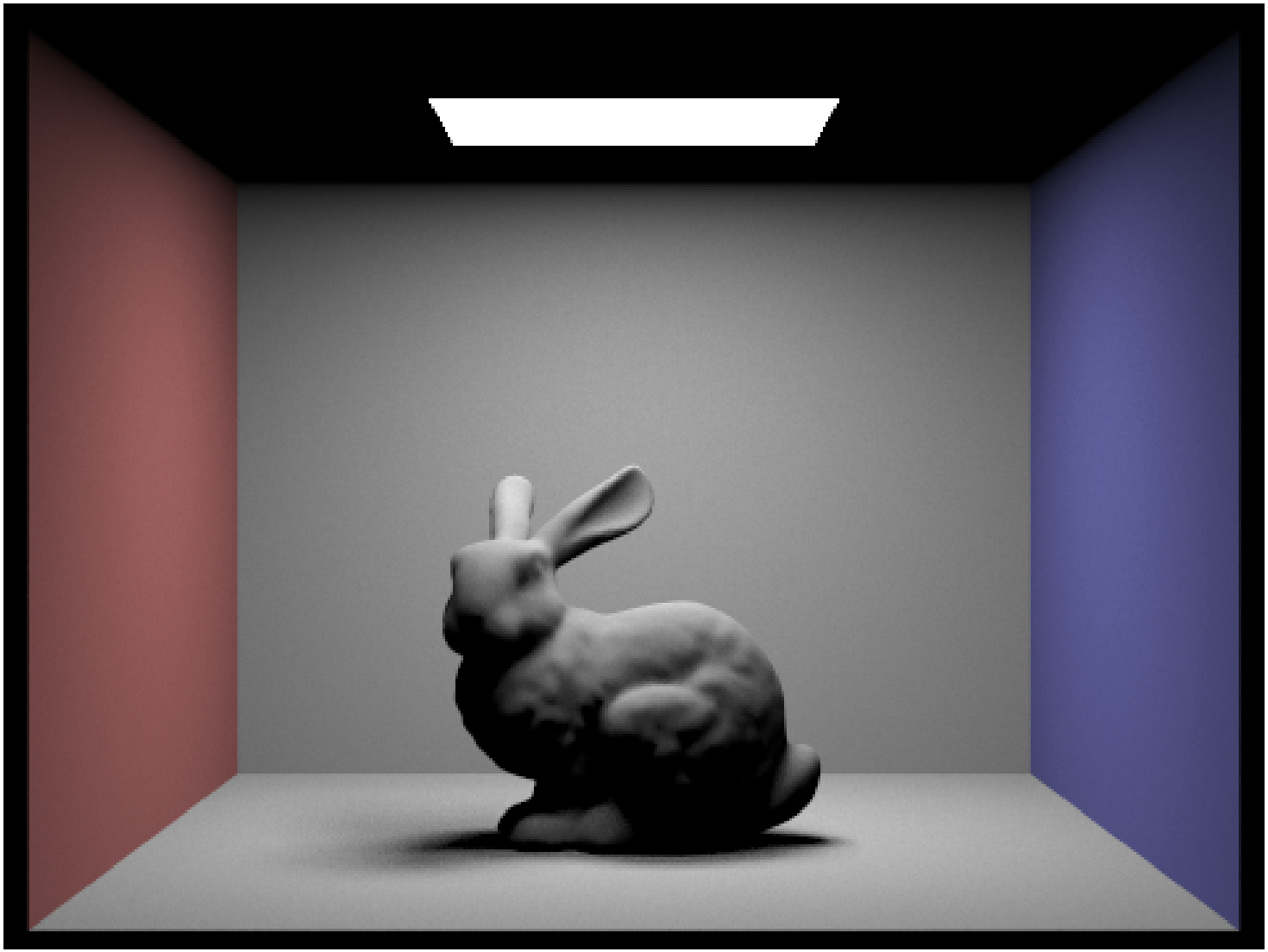

Figure 24: 1024 samples per pixel, 1 samples per area light, direct lighting with importance sampling.

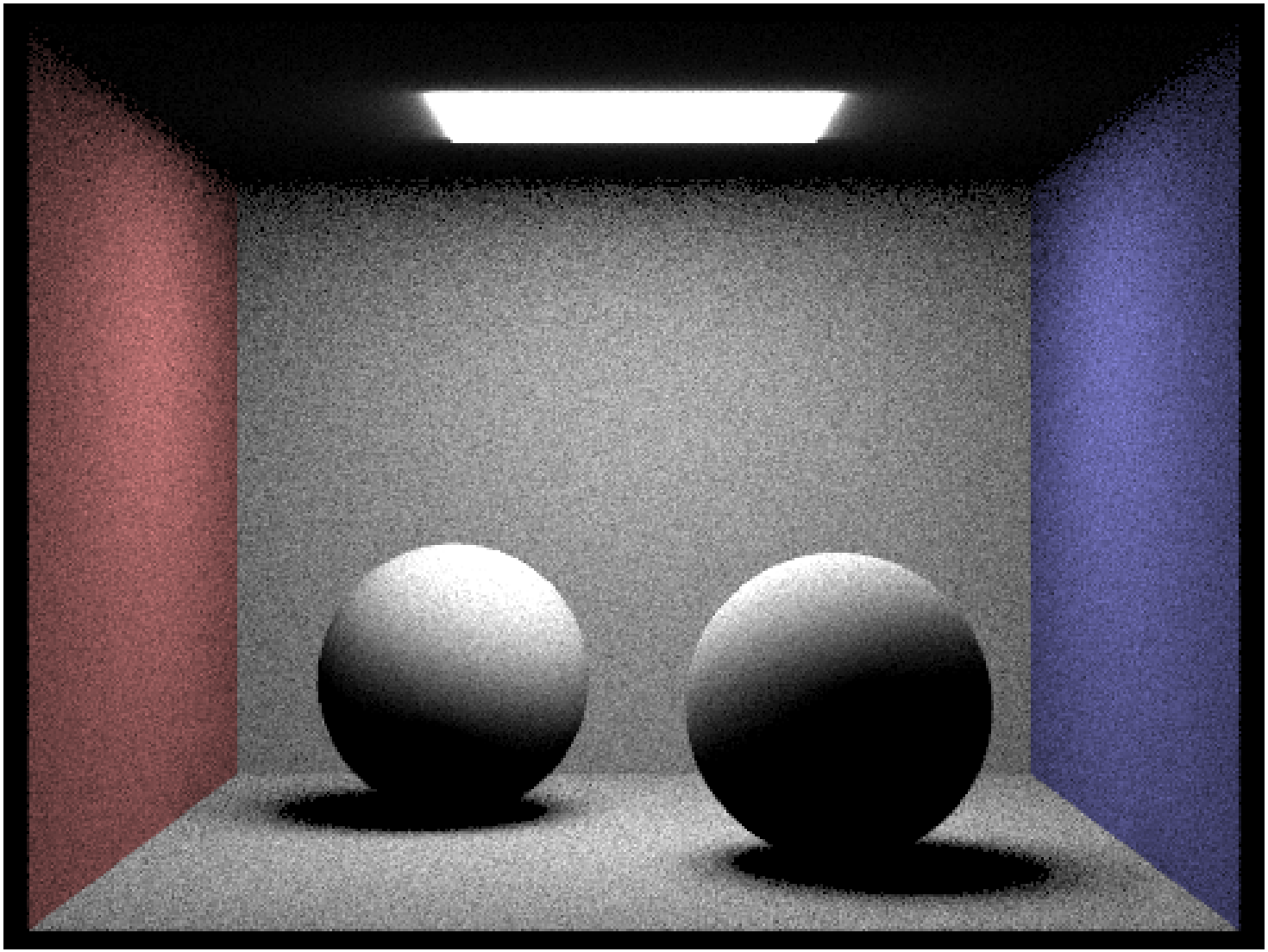

Figure 25: 1024 samples per pixel, 1 samples per area light, direct lighting with uniform hemisphere sampling.

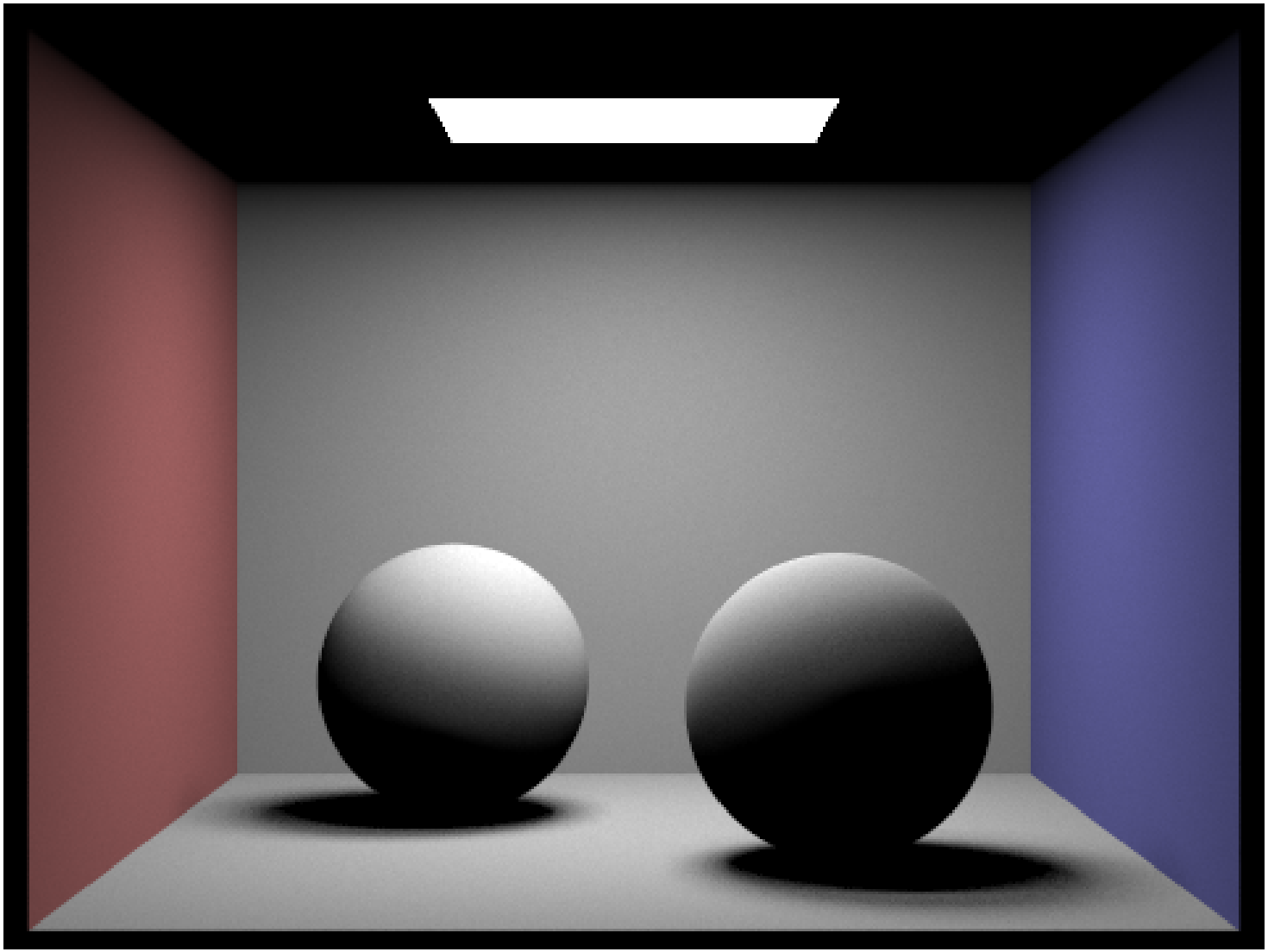

Figure 26: 1024 samples per pixel, 1 samples per area light, direct lighting with importance sampling.

Below we have included pictures of two spheres in a Cornell box rendered with 1, 4, 16, 64 light rays with importance sampling.

As we can see, when we only take 1 sample per pixel, the resulting pictures are relatively noisy even as the number of samples per area light increases. The noise mainly takes the form of black pixels spreading across whole images, and is most pronounced when we only take 1 sample per area light. The resulting black pixels show a large variance in the samples taken. This result is to be expected since we are relying on only very few terms in the Monte Carlo Integration and the resulting pictures will be heavily biased towards those samples taken. Even though the noise decrease as the number of samples per area light increases, we can see that the the edges of the spheres are less well-defined compared to pictures generated with more samples per pixel.

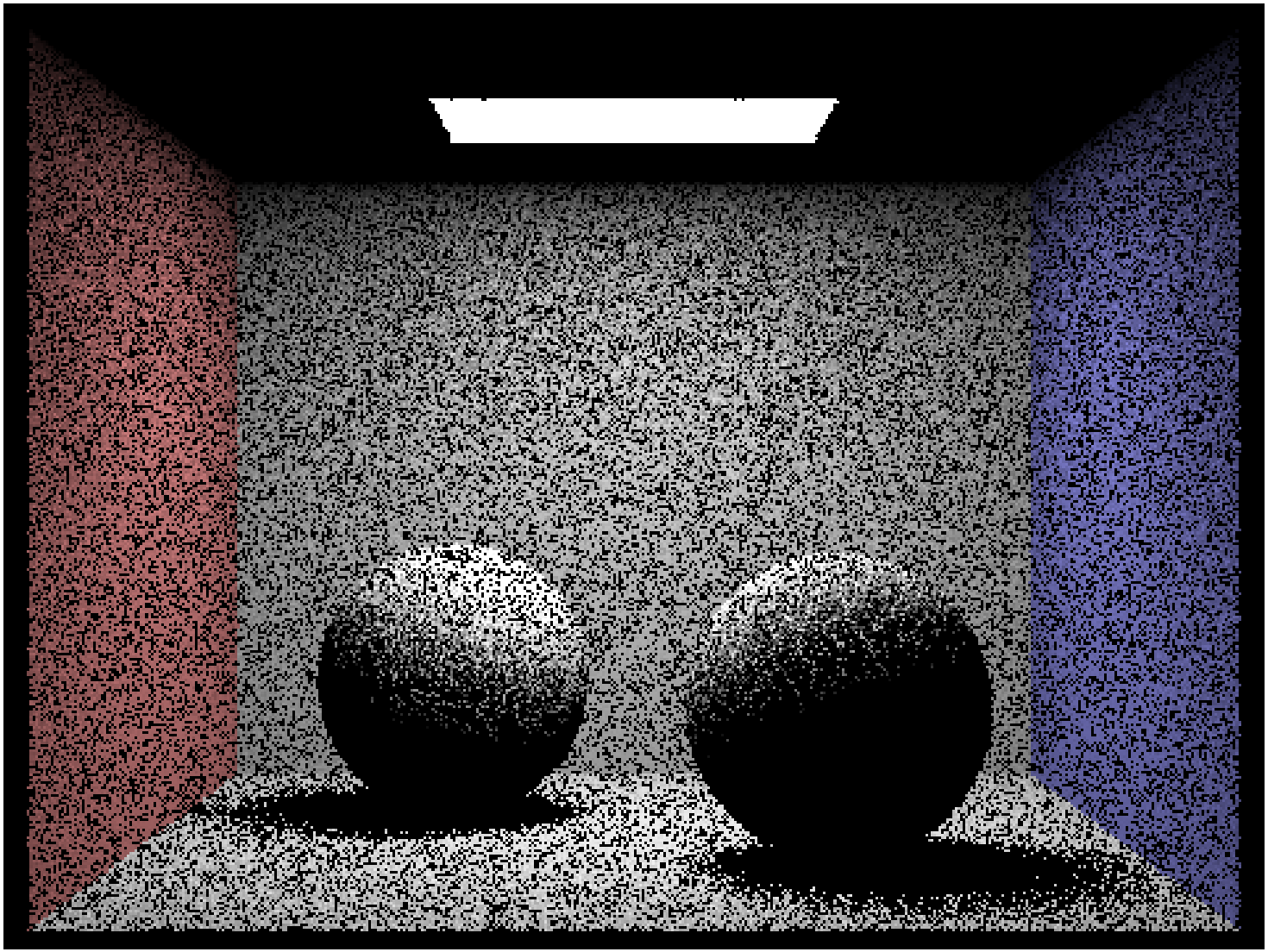

Figure 27: 1 samples per pixel, 1 samples per area light, direct lighting with importance sampling.

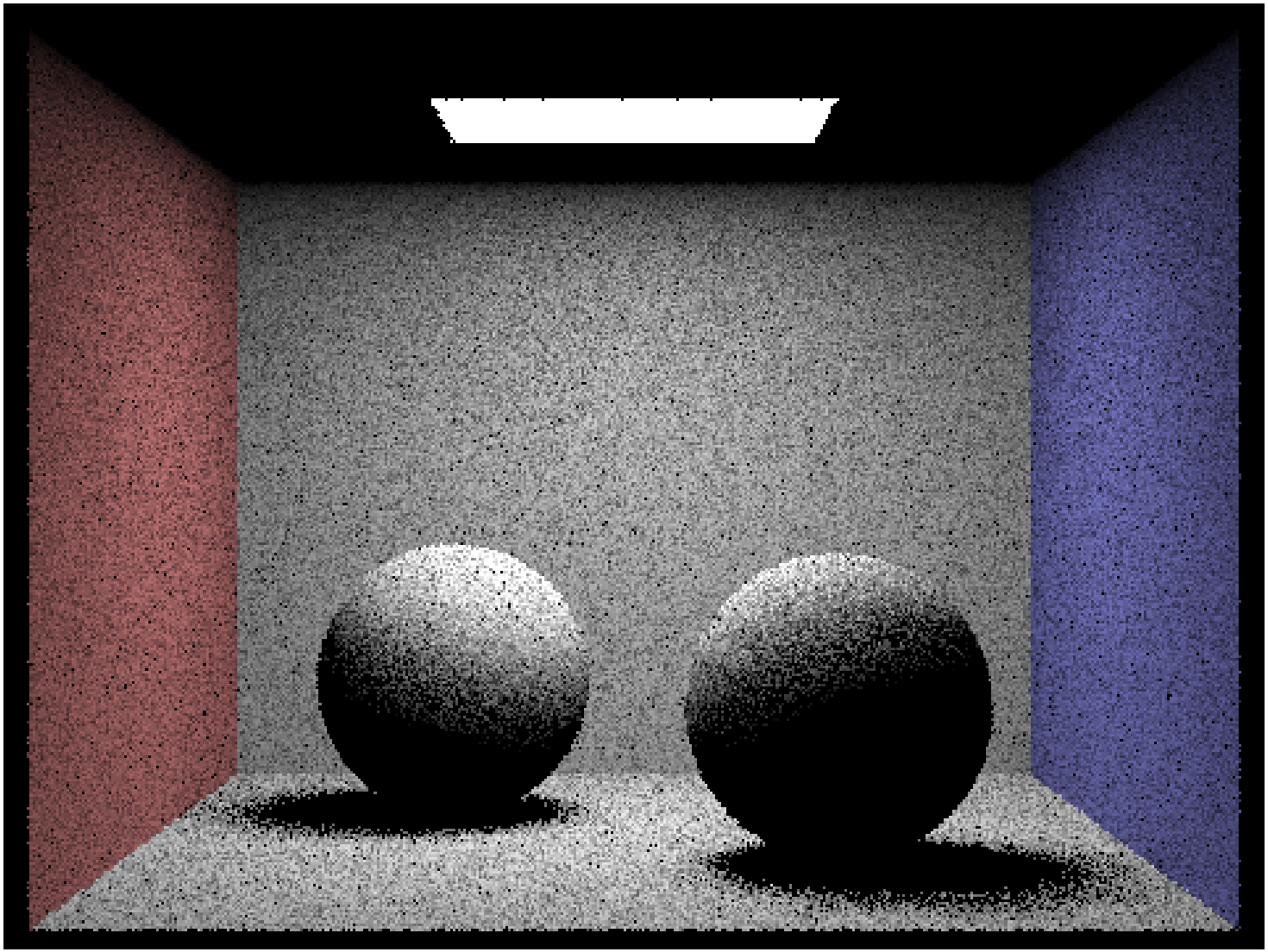

Figure 28: 1 samples per pixel, 4 samples per area light, direct lighting with importance sampling.

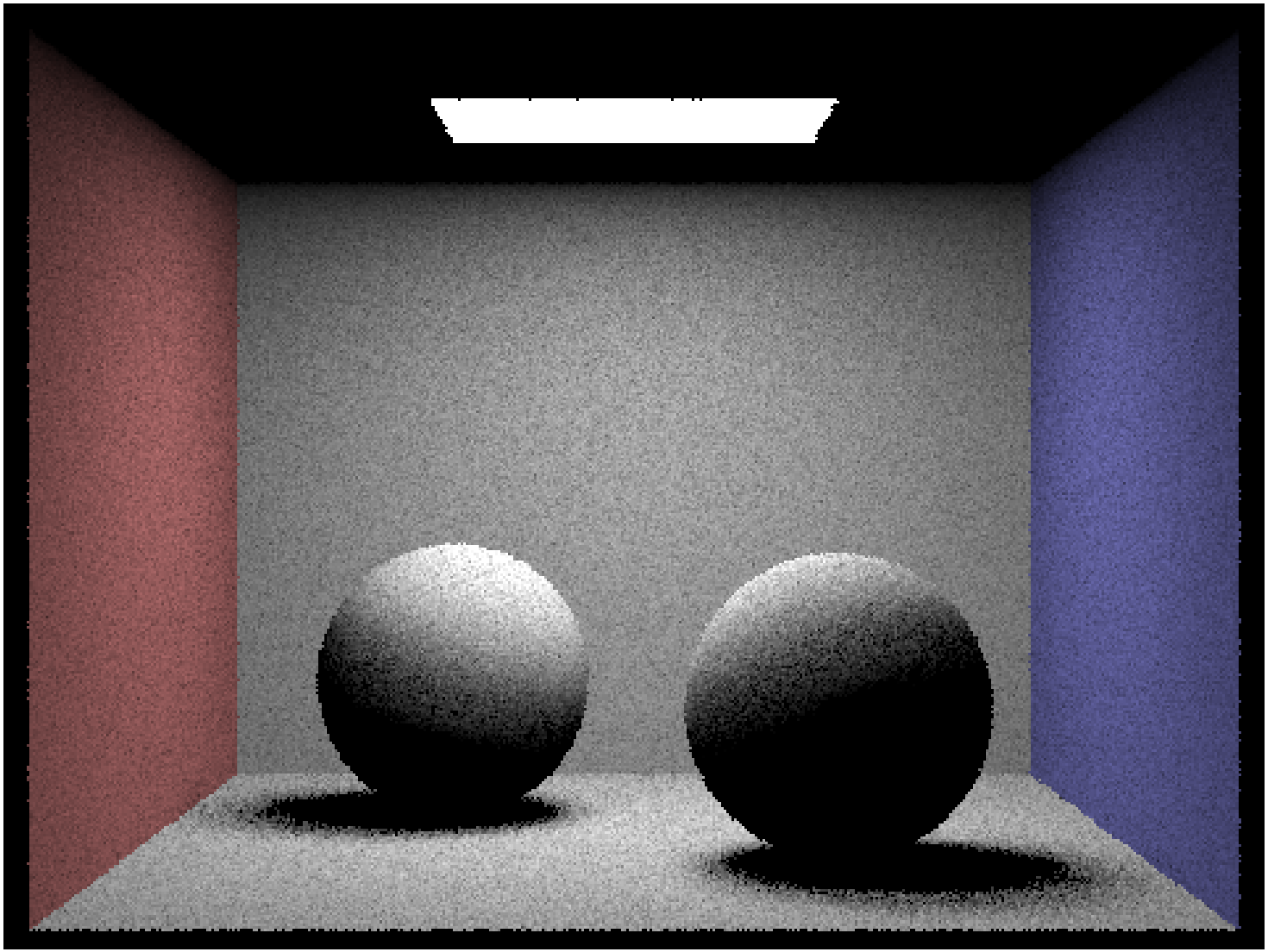

Figure 29: 1 samples per pixel, 16 samples per area light, direct lighting with importance sampling.

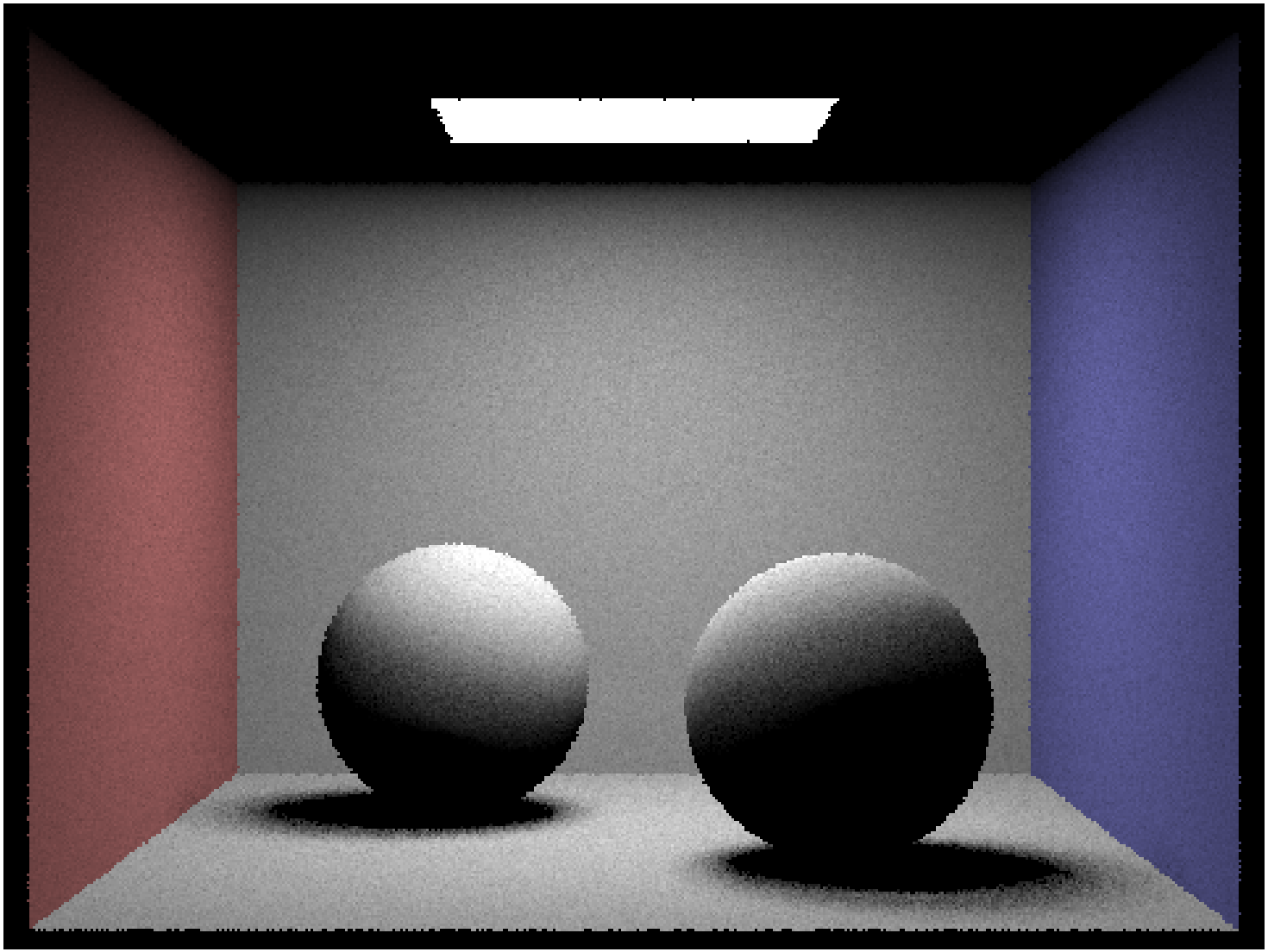

Figure 30: 1 samples per pixel, 64 samples per area light, direct lighting with importance sampling.

4 Global Illumination

In Section 3, we simplified to ray-tracing problems to only the light source and the direct bounce from the light source. Because much of the energy of later reflections are not considered, the pictures rendered usually appear dim. When implementing the global illumination, we try to incorporate these indirect bounces into our algorithm to simulate the light transport better.

We can separate the the radiance at one point to direct lighting, which we have computed in Section 3 and indirect lighting. However, one immediate challenge of computing the indirect lighting is that it requires the indirect lighting term from the previous bounce, resulting in a natural recursive structure. If the indirect lighting is the only component we need to consider, then by the natural decay with each light reflection, it is possible to prove the convergence with ϵ-δ reasoning. However, we also need to consider the direct lighting component of the previous bounce to get the correct radiance. Therefore, even if we arbitrarily increase the depth of the reflection in our Monte Carlo estimation, we cannot prove that the sequence of the integrands will converge. Arbitrarily terminating at depth $N \in \mathbb{Z}_+$ will introduce a bias in the generated picture. We will first implement this biased version of global illumination by terminating rays at the maximum depth.

Since our surface BSDF is diffuse and light can come from all direct directions, with each direction requiring a new recursive function call, we have to sample the incoming angles and apply Monte Carlo Integration to avoid infinitely many function calls. We use the `sampler` field to get a sample of `wi` and its probability `pdf` according to the cosine weighted hemisphere distribution with the `sample_f` function. The function will also return the BSDF value evaluated at `wo` and `wi`, which is $\frac{\rho}{\pi}$ for diffuse BSDF.

For each intersection of the ray with the surface, we let `L_direct` to be the direct illumination calculated with `one_bounce_radiance` function in Section 3. In our C++ program, we will use the increment scenario where the ray starts at depth 0. In the problem specification, if the `max_ray_depth` field is 1, then we only return the direct lighting. Since `one_bounce_radiance` does not increase ray depth and the a ray of depth 1 in our scenario is the first indirect bounce, we will terminate further recursions when the depth of the ray is equal to `maximum_ray_depth - 1`. Our function `at_least_one_bounce_radiance` will combine the radiance from direct illumination and the indirect illumination and output `L_out = L_direct + L_indirect`. which is `L_direct + L_indirect`. For the debug mode where `isAccumBounces` is false, we need to change the `est_radiance_global_illumination` function such that it correctly outputs `zero_bounce_radiance` when the maximum ray depth is 0, `one_bounce_radiance` when the maximum ray depth is 1, and `at_least_one_bounce_radiance` when the maximum ray depth is greater than 1. In our `at_least_one_bounce_radiance` function, we also need to make sure only `L_direct` is returned for the last bounce (where the scene is directly illuminated) and only `L_indirect` is returned for the previous bounces (intermediate bounces before the light reaches the viewer).

From the current intersection, we sample the incoming direction `omega_i`,

its probability `pdf`, and the BSDF value `sb` with `sample_f`. Similar to our discussion in Section 3, `omega_i.z` will be the cosine term in Lambert's Cosine Law. If we get an invalid sample where the probability is very close to zero or the cosine term is less than zero, then we return `L_out` immediately. We can construct a ray that has 1 more depth of the current ray with `Ray rb = Ray(hit_p, o2w * omega_i)`. We also set the `min_t` field to be `EPS_F` to avoid self intersection. Then we can use our BVH data structure to compute the intersection `ib`. If `ib` exists, then the incoming `Li` from the intersection `ib` at the current intersection is `at_least_one_bounce_radiance(rb, ib)`. Then we can use the Monte Carlo estimator to calculate the estimated `L_indirect` term with `sb * Li * omega_i.z / pdf`.

However, there are two major flaws with the current implementation. The first is that if the sampled `omega_i` comes from the light source, we will be double counting the radiance from the light source. Therefore, we have to add an additional check at the beginning of the function `at_least_one_bounce_radiance` to return zero when the intersection has non-zero emission (the light emission of light sources has been handled in `zero_bounce_radiance`). Another issue is that when the `maximum_ray_depth` is 0, we want to output only the zero bounce radiance, so we have to output only that term in `est_radiance_global_illumination` function for zero ray depth.

As we have discussed earlier, terminating Monte Carlo Integration at an arbitrary maximum ray depth introduces a bias. To avoid such artificial bias for an infinite process, we introduce the probabilistic termination criteria with Russian Roulette. We compute at least one indirect bounce and afterwards each bounce has a 0.35 chance to terminate. In our implementation, we set the continue probability `double cpdf = r.depth < 2 ? 1.0 : 0.65` and use the `coin_flip` function to determine whether the recursion should continue. We compensate for the early termination, we also have to divide our `L_indirect` term by `cpdf`. Note the first indirect bounce is guaranteed to happen, the term `cpdf` we used to divide `L_indirect` is correctly set to 1. In addition, we want to be able to turn off the indirect bounce radiance by setting `max_ray_depth` to 1, so we change `est_radiance_global_illumination` to return only direct lighting when the `max_ray_depth` is 1.

Below we have included some pictures rendered with global illumination.

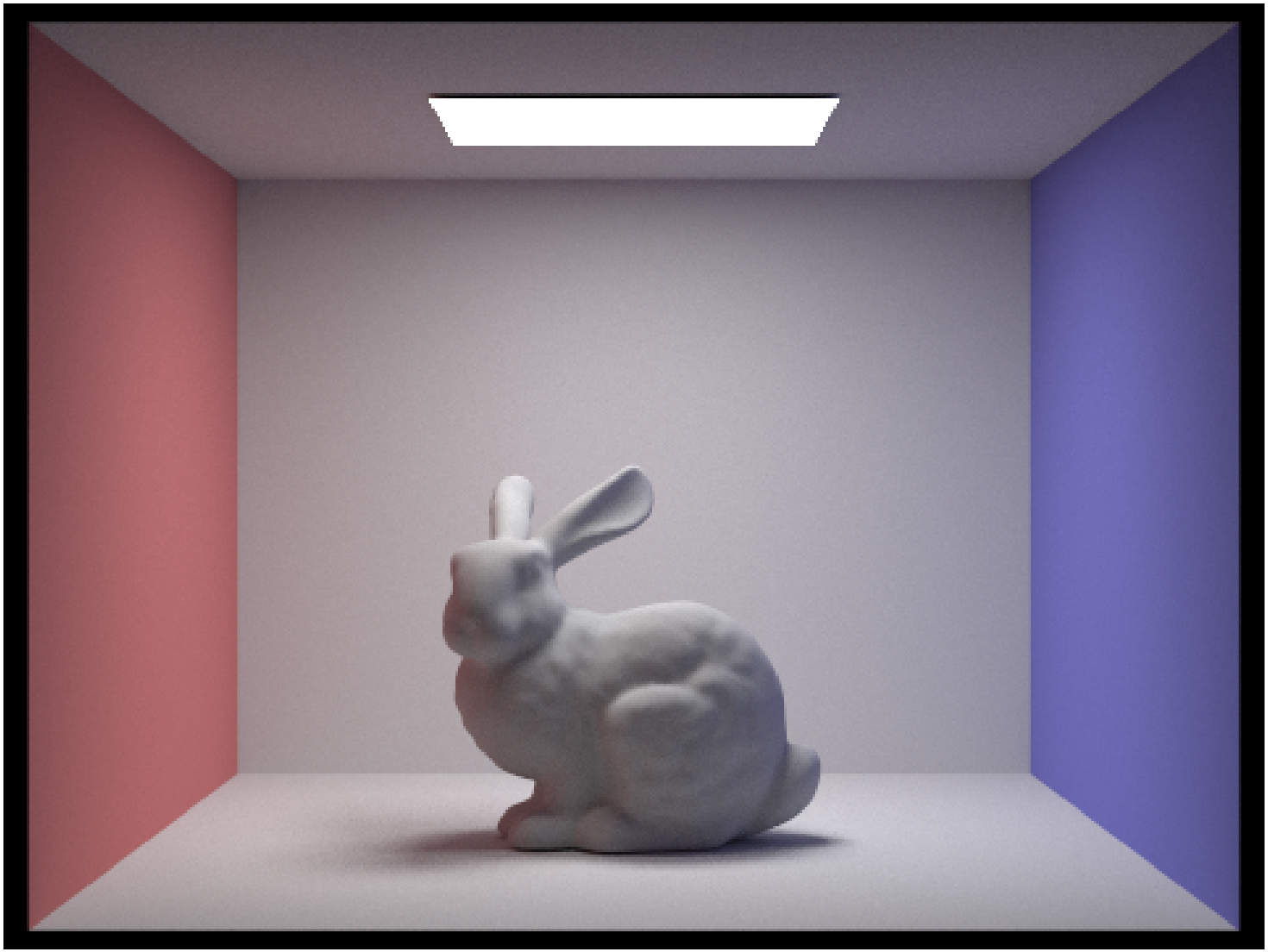

Figure 31: 1024 samples per pixel, 1 samples per area light, max ray depth 5.

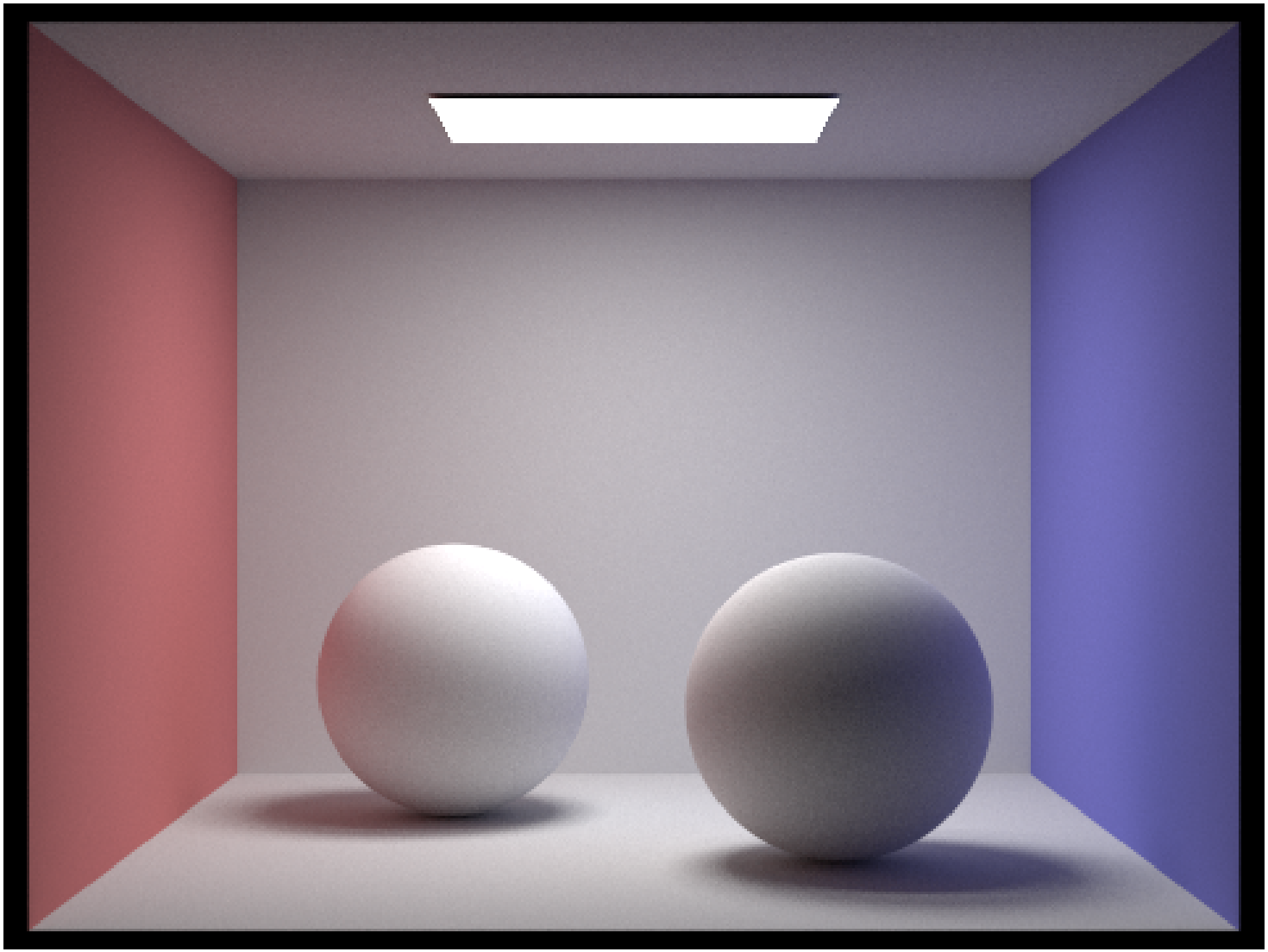

Figure 32: 1024 samples per pixel, 1 samples per area light, max ray depth 5.

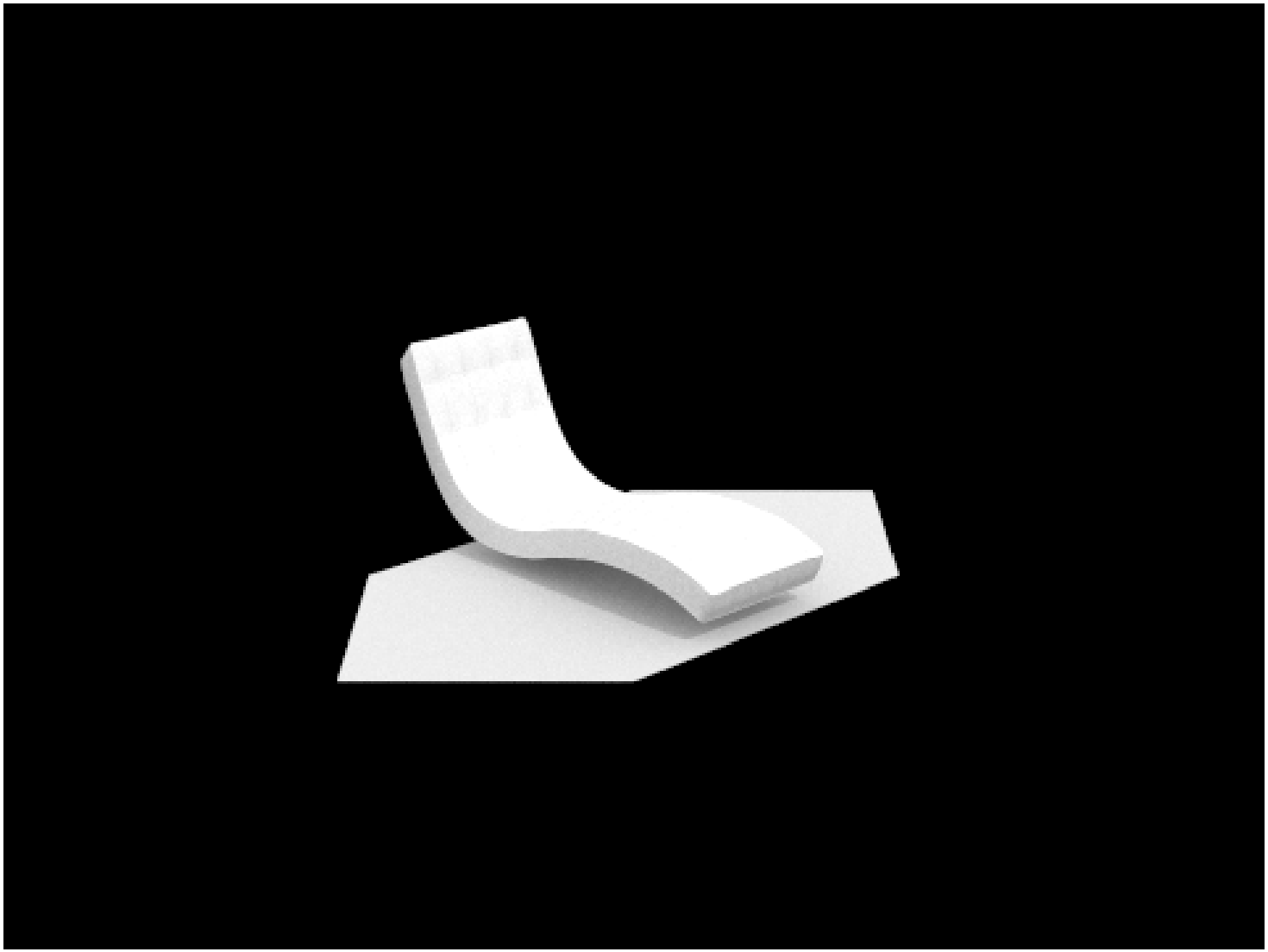

Figure 33: 1024 samples per pixel, 1 samples per area light, max ray depth 5.

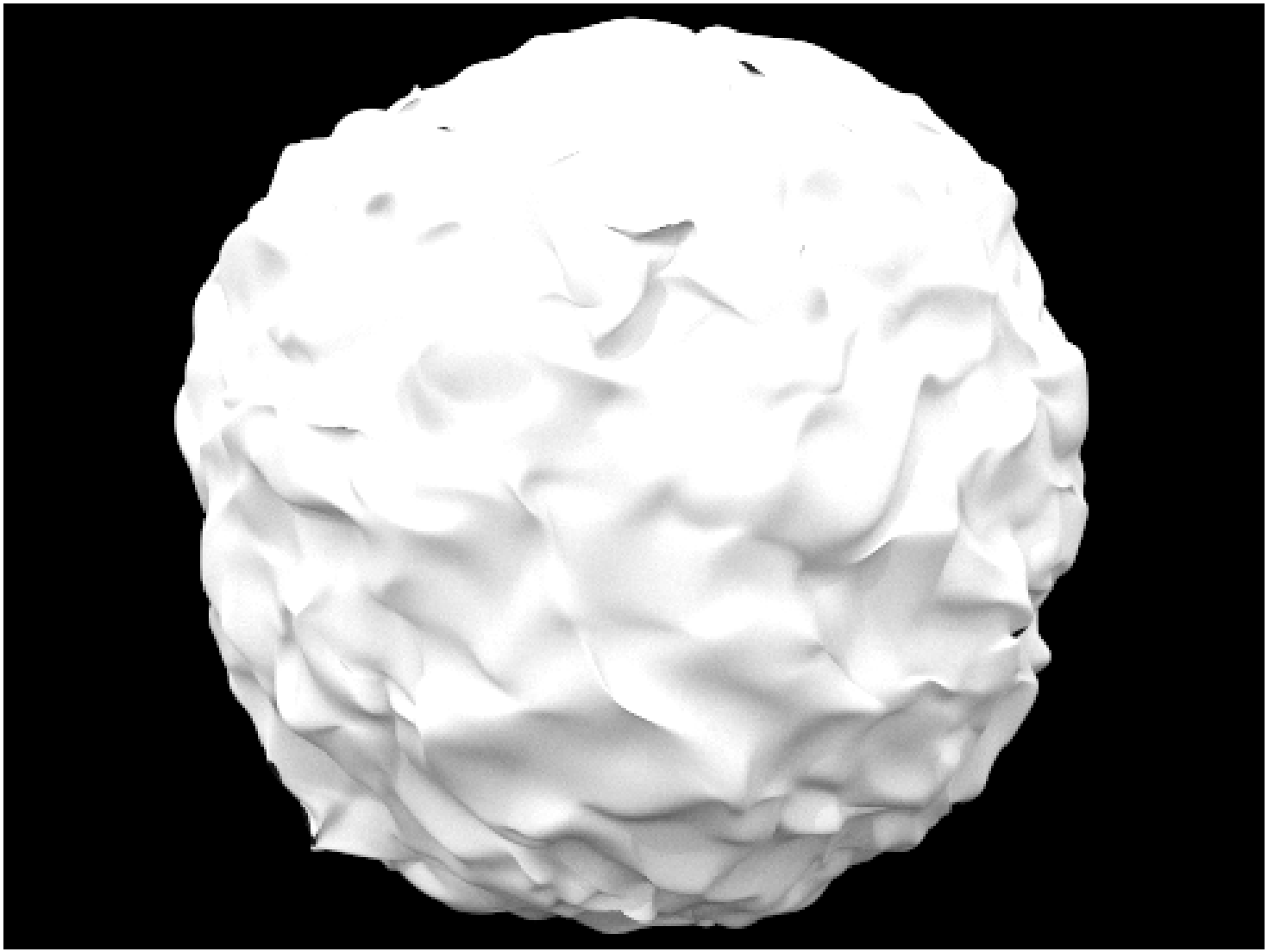

Figure 34: 1024 samples per pixel, 1 samples per area light, max ray depth 5.

Figure 35: 1024 samples per pixel, 1 samples per area light, max ray depth 5.

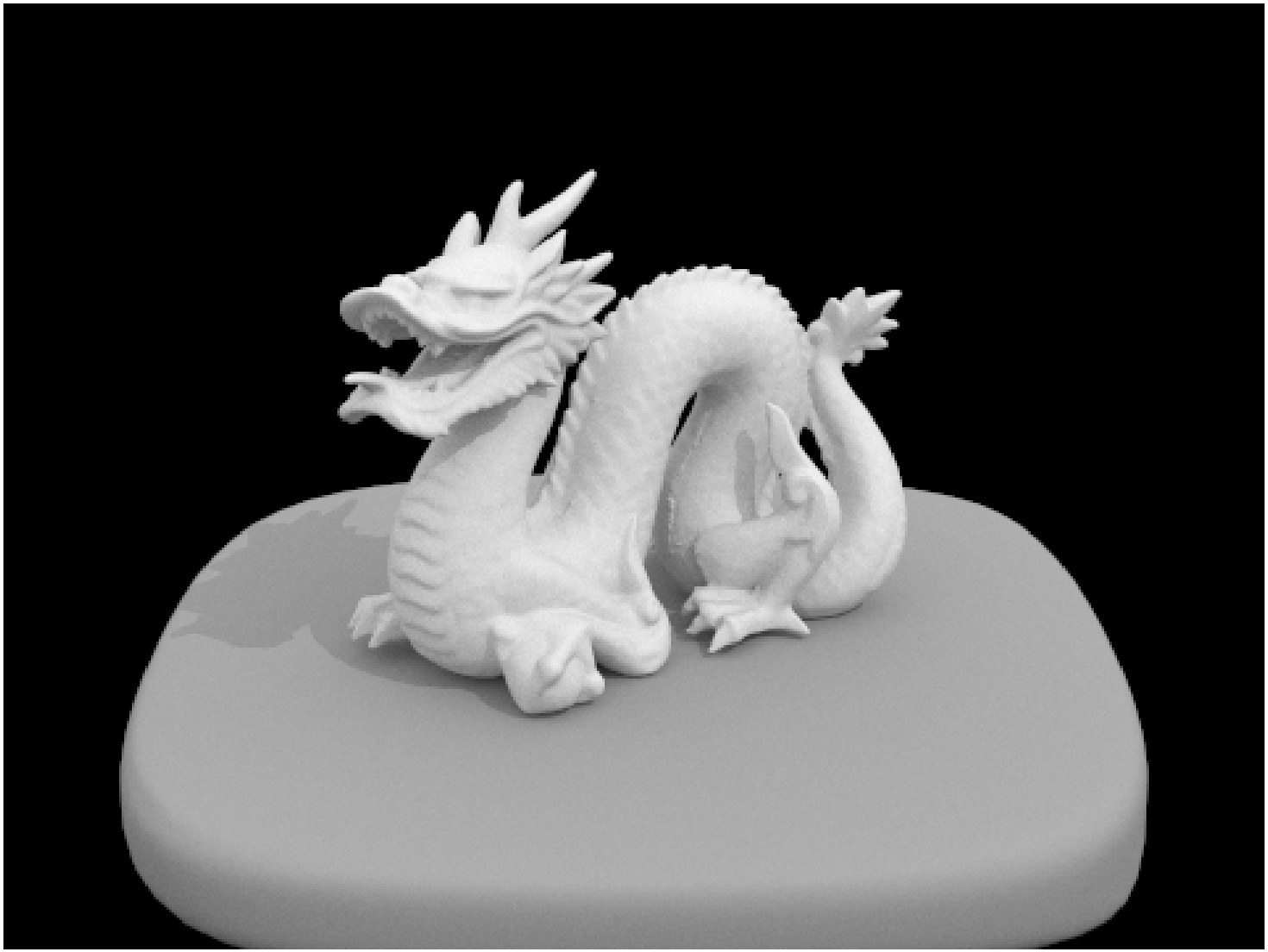

Figure 36: 1024 samples per pixel, 1 samples per area light, max ray depth 5.

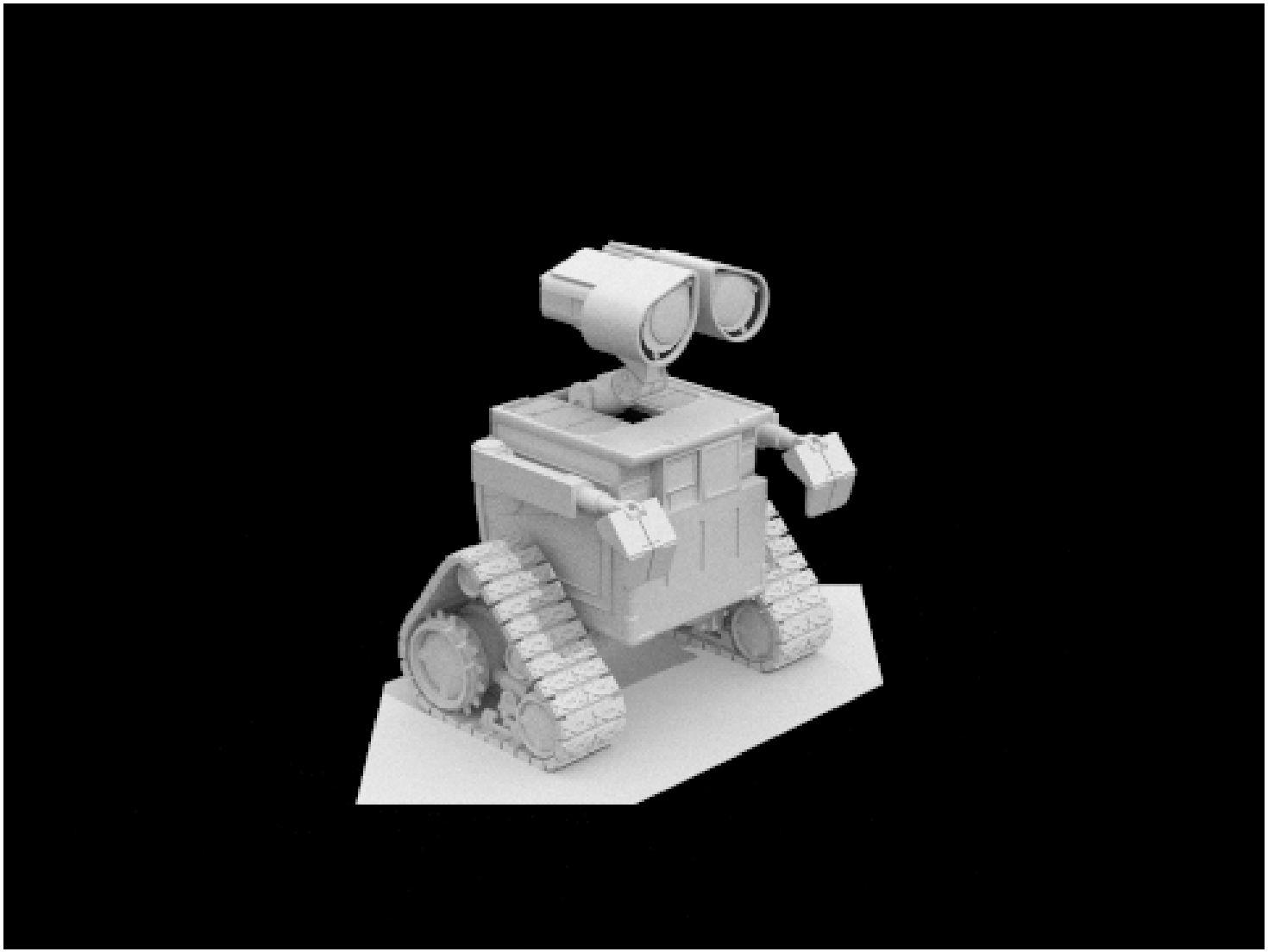

Figure 37: 1024 samples per pixel, 1 samples per area light, max ray depth 5.

Below we have included pictures of two spheres inside a Cornell box with only direct illumination and with only indirect illumination.

As we can see, direct lumination creates high contrast images where the regions hit by the light are bright while the regions not hit by the light are dark. The shadows created by direct lumination are also completely black. In comparison, the picture of only indirect lumination appears a bit dimmer due to the energy decay in the light transport. The color transition also appears much softer. In addition, the shadows are not completely black. Adding indirect illumination into global illumination softens the color transitions in the scene and create shadows with realistic color.

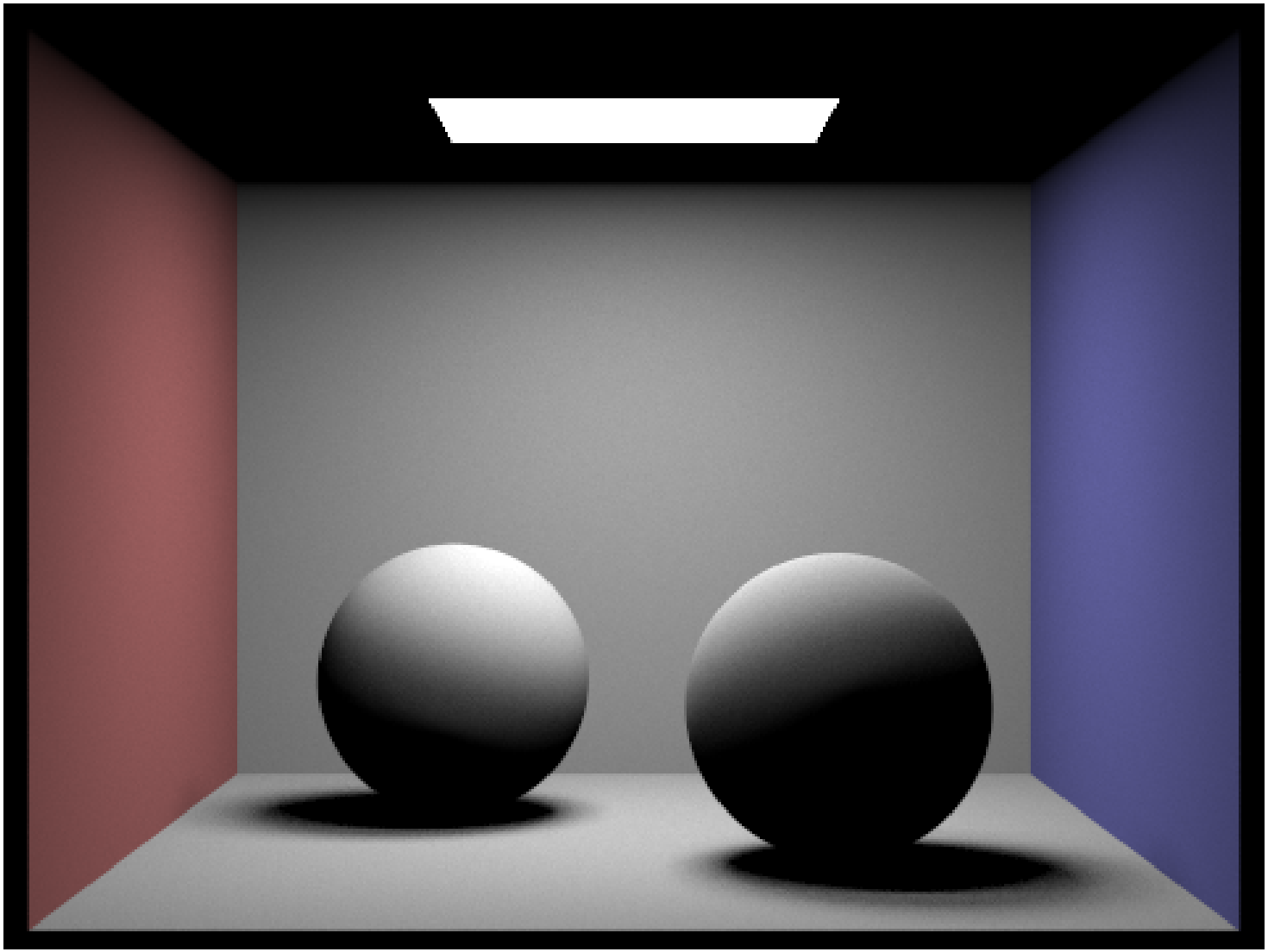

Figure 38: 1024 samples per pixel, 1 samples per area light, max ray depth 5, direct illumination only.

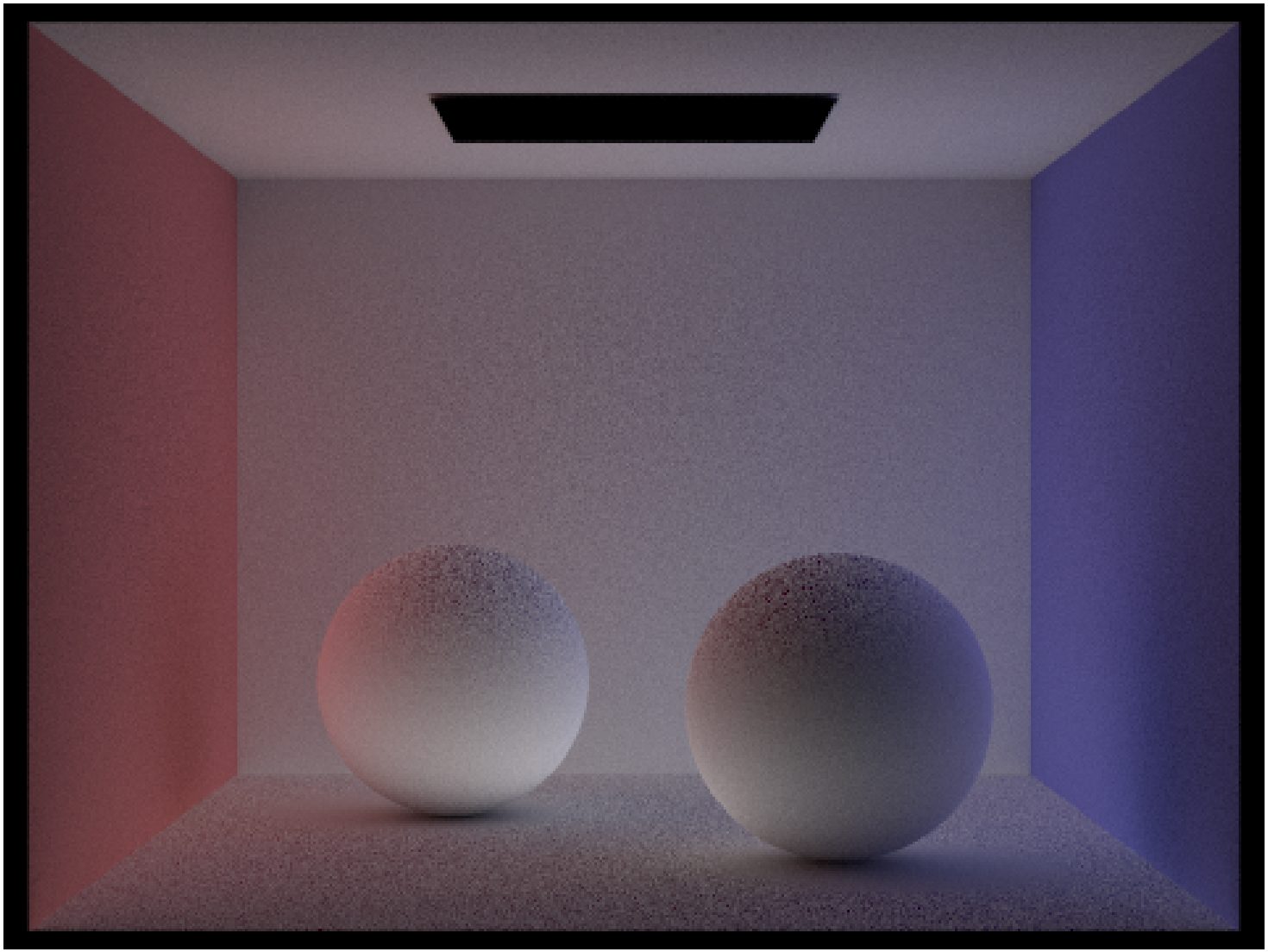

Figure 39: 1024 samples per pixel, 1 samples per area light, max ray depth 5, indirect illumination only.

Below we have included pictures of a bunny inside a Cornell box with `max_ray_depth` set to 0, 1, 2, 3, 4, and 5 for both when `isAccumBounces` is true and false.

We can see that at the second bounce of the light (the first indirect bounce), the picture is much dimmer than the direct lighting due to the energy decay in light transport. The upper half of the bunny that was bright in the direct illumination becomes relatively dark in this light bounce. The bottom half of the room, the lower half of the bunny, and the ceiling except the light source are relatively bright compared to the rest of the picture. We can also see that the shadow appears much softer in this light bounce, and blends into the environment well. Adding this light bounce to the rendered picture softens the color transition and illuminates the lower part of the bunny, the bottom part of the room, and the ceiling that were dark in the direct illumination. The contrast of the shadow with the floor is also less pronounced. In addition, due to the accumulation of the illumination of multiple light bounces, the picture generated becomes a bit brighter.

At the third bounce the overall brightness of the picture diminishes sharply. The brightness of the room appears much more uniform except the two corners that appear slightly brighter than the rest of the room. The bunny is much darker than the rest of the room, and the shadow of the bunny is also soft. Adding this light bounce to the rendered picture further brightens up the scene, especially the room, and makes the color transitions even more natural.

The second and third bounces of light in the global illumination simulate how light behaves in physics under the assumption of geometric photometry. The soft color transitions and the shadows are generated by combining multiple physically accurate light bounces. The natural transition of the color and the physically plausible shadows are hard to replicate in rasterization, where only the primitive with the lowest z-buffer at a given pixel is rendered. Even though rasterization can achieve some shadow effects and soft color transitioning with finer mesh modeling and texture mapping, these techniques rely on linear interpolation, resulting in unnatural patterns when trying to imitate the largely non-linear color transitions and shadows in real world. Unlike rasterization that discards the information about the primitives that are blocked by other objects in the front, path-tracing considers how light transports across all primitives in the scene, even if they appear behind some other objects. For example, the wall behind the bunny still has its BSDF and contributes to the radiance of multiple light bounces.

We can see that the pictures accumulating multiple stages of light bounces become largely stable after the first indirect bounce, and the later pictures just become a little bit softer. The pictures not accumulating multiple light bounces show large variances between pictures but become significantly dimmer due to the energy loss in light transport, so they contribute less in the final rendered picture.

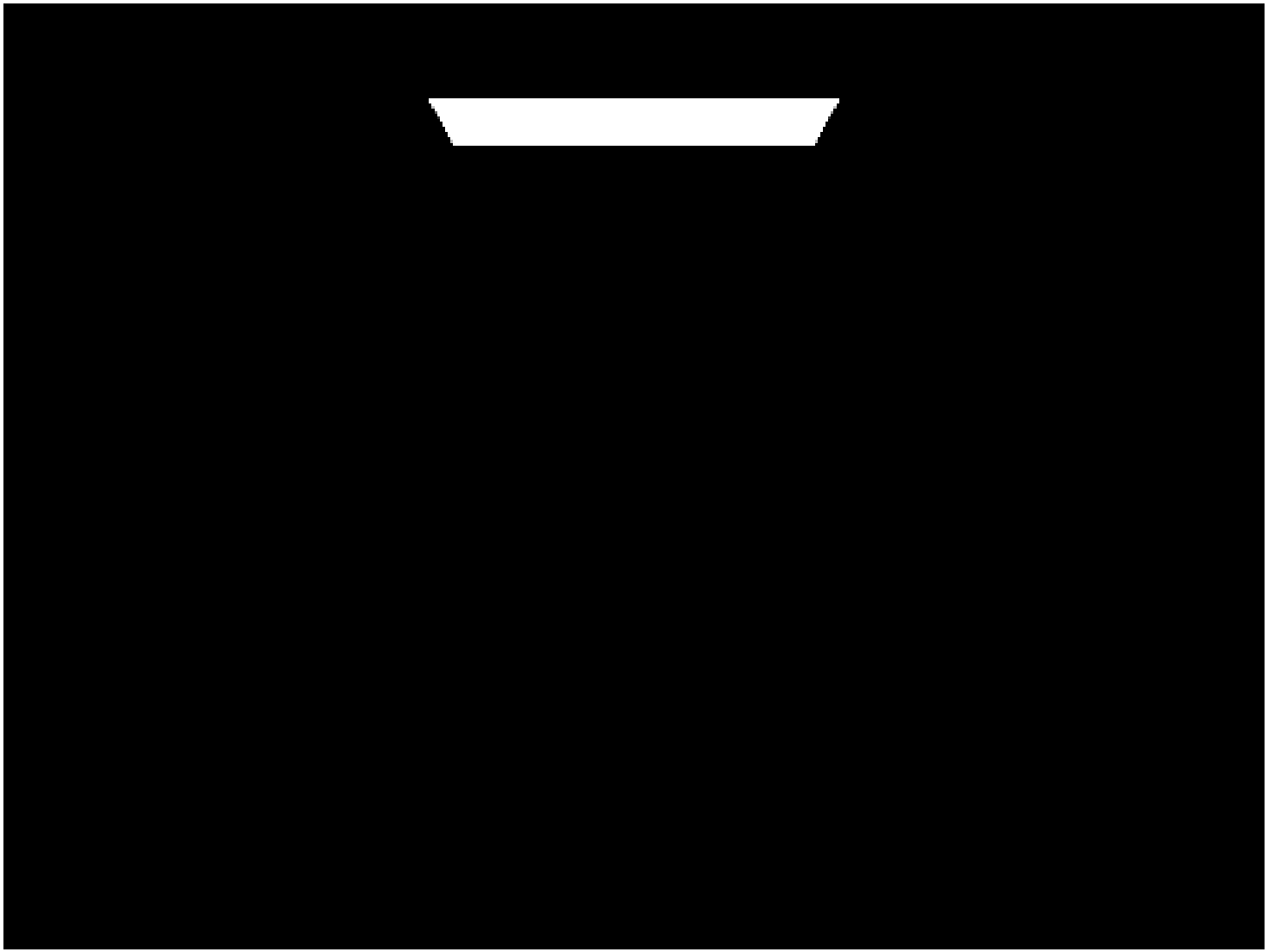

Figure 40: 1024 samples per pixel, 1 samples per area light, max ray depth 0, `isAccumBounces` true.

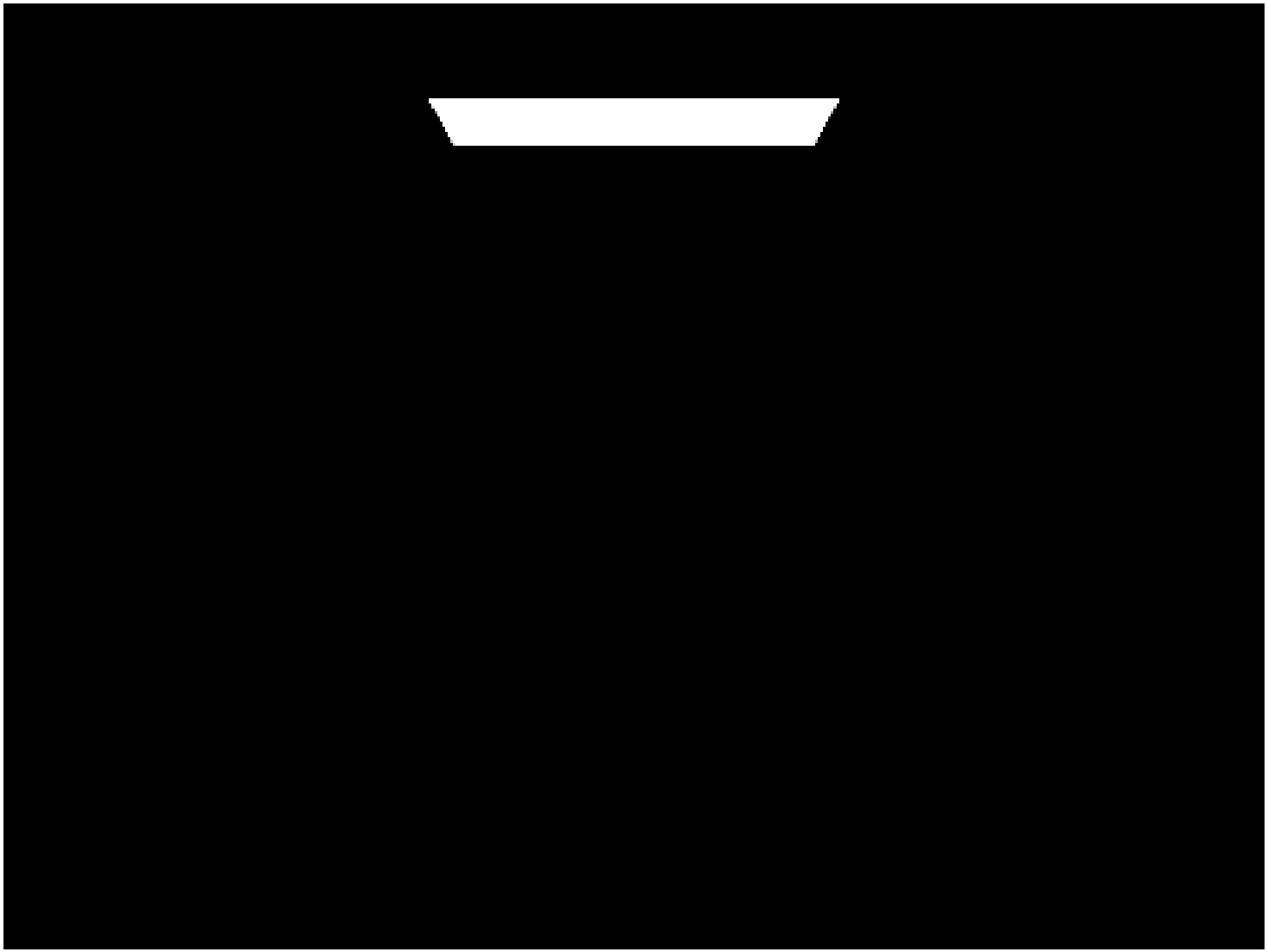

Figure 41: 1024 samples per pixel, 1 samples per area light, max ray depth 0, `isAccumBounces` false.

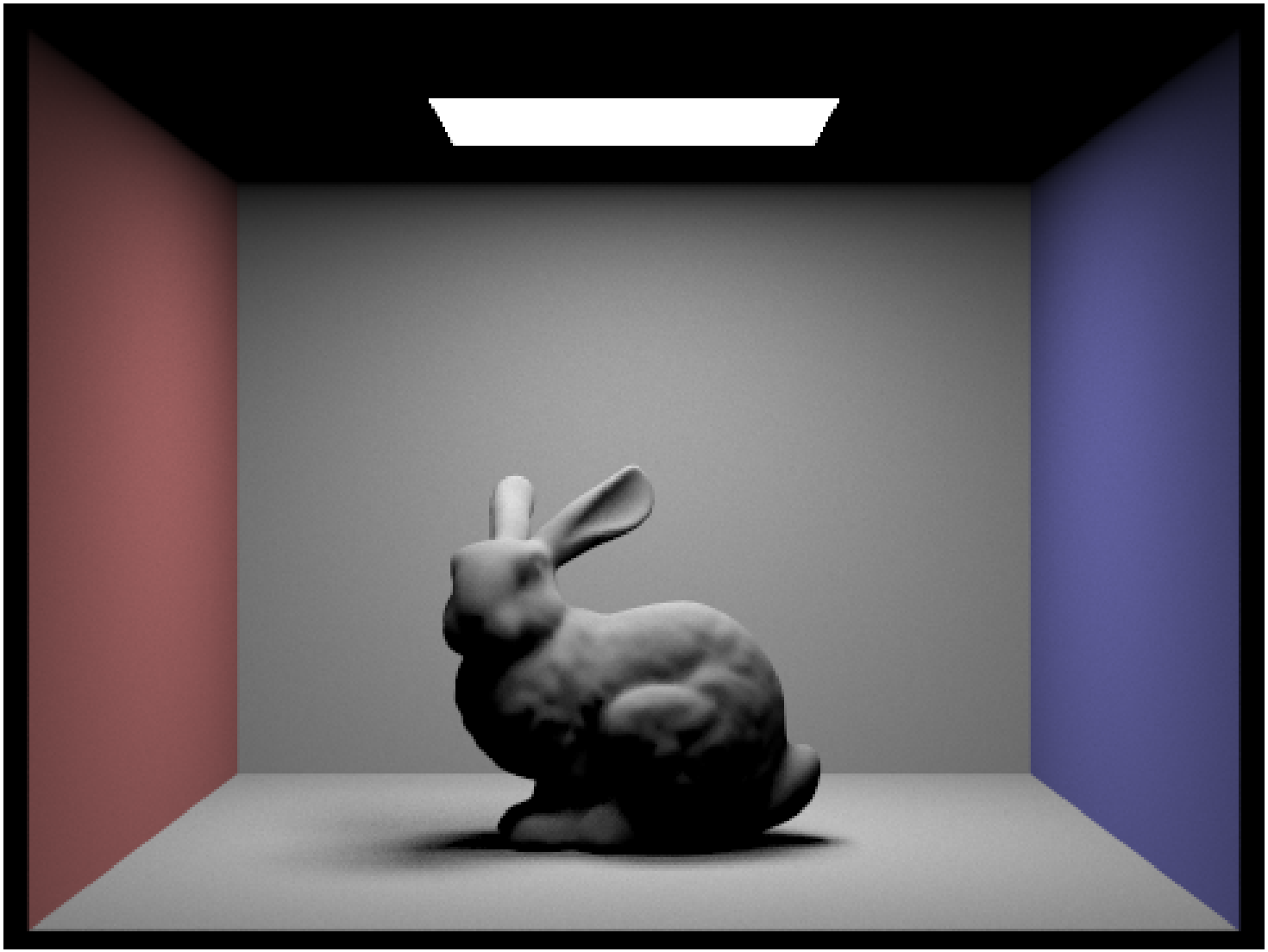

Figure 42: 1024 samples per pixel, 1 samples per area light, max ray depth 1, `isAccumBounces` true.

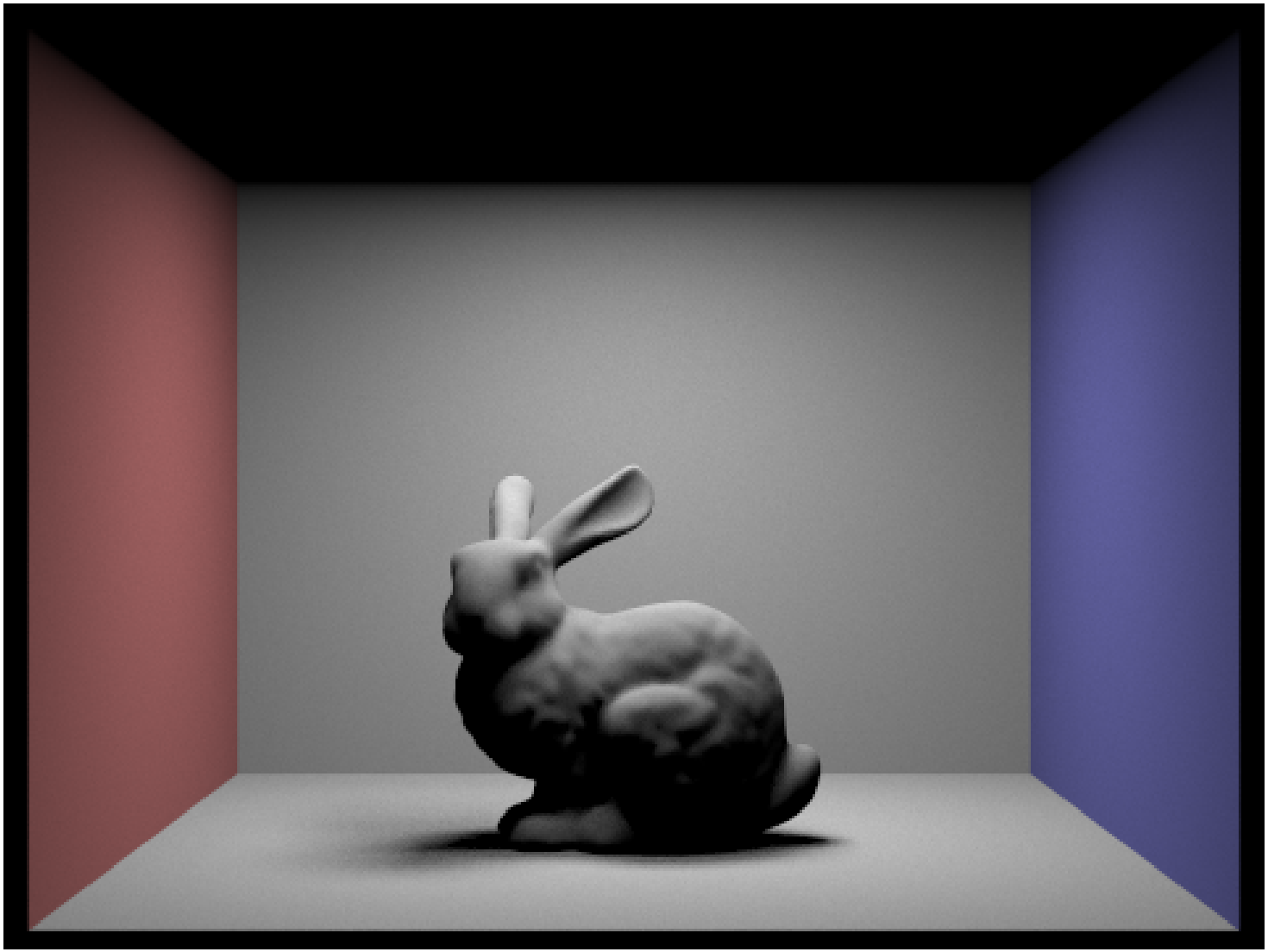

Figure 43: 1024 samples per pixel, 1 samples per area light, max ray depth 1, `isAccumBounces` false.

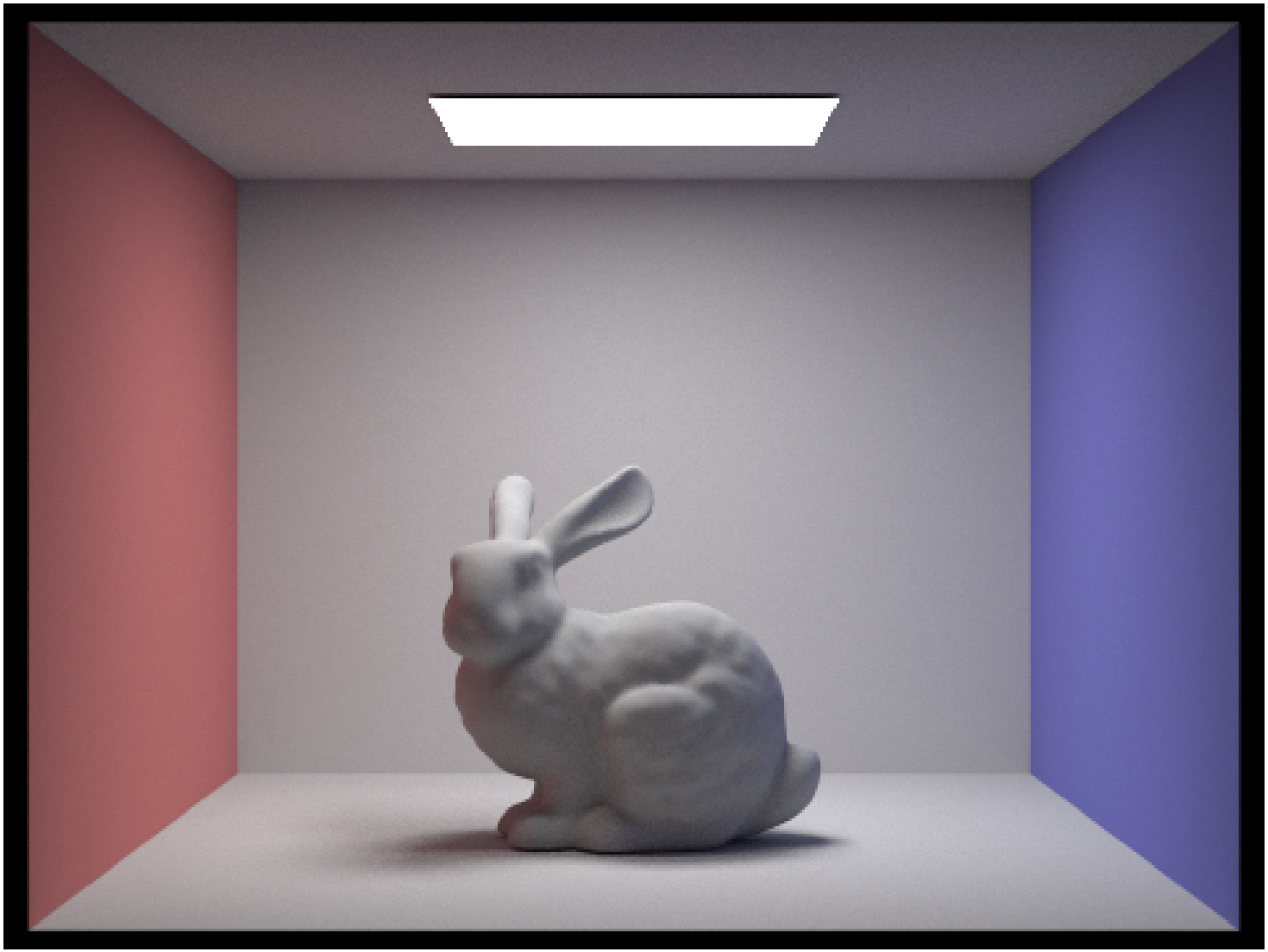

Figure 44: 1024 samples per pixel, 1 samples per area light, max ray depth 2, `isAccumBounces` true.

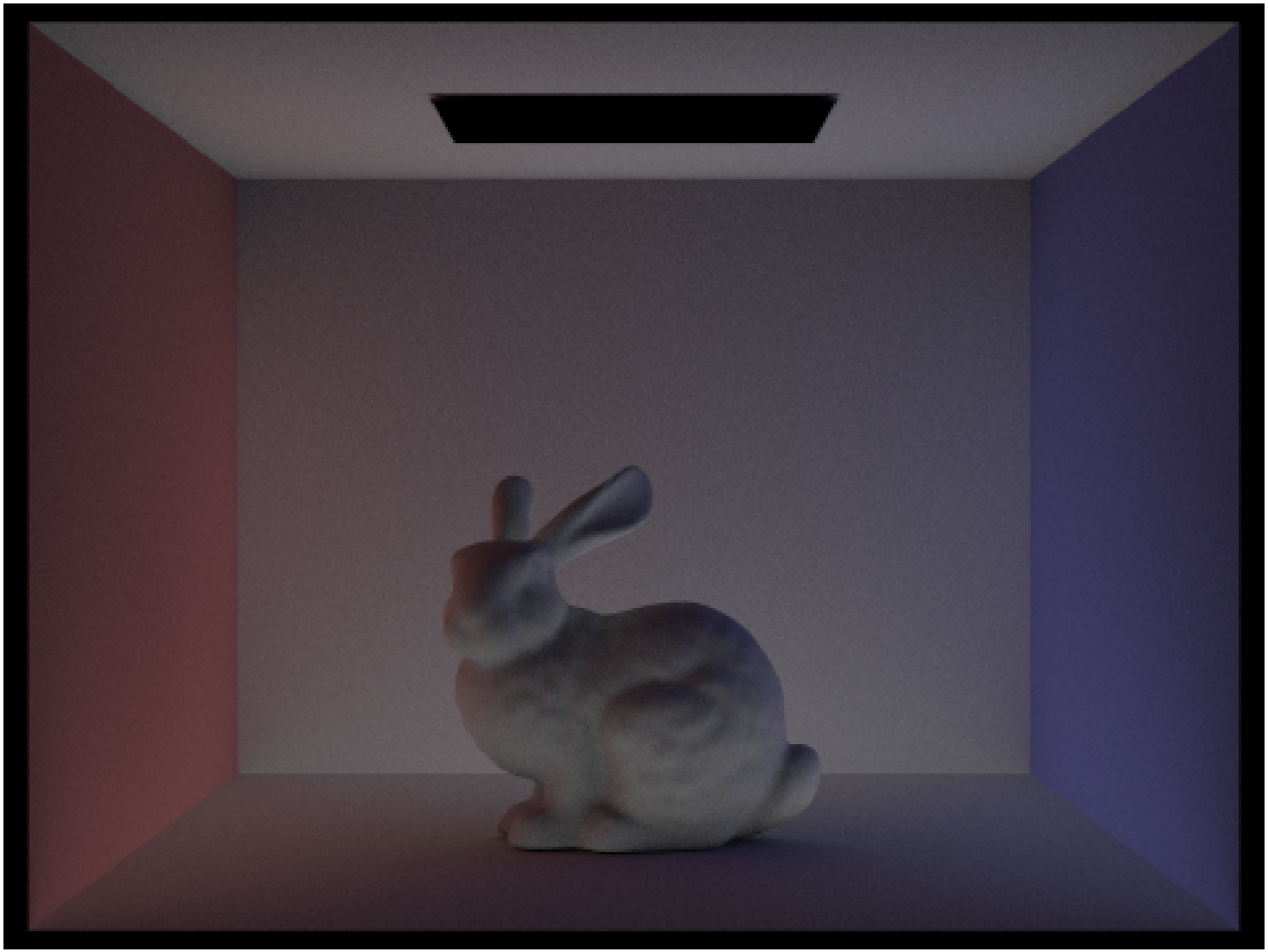

Figure 45: 1024 samples per pixel, 1 samples per area light, max ray depth 2, `isAccumBounces` false.

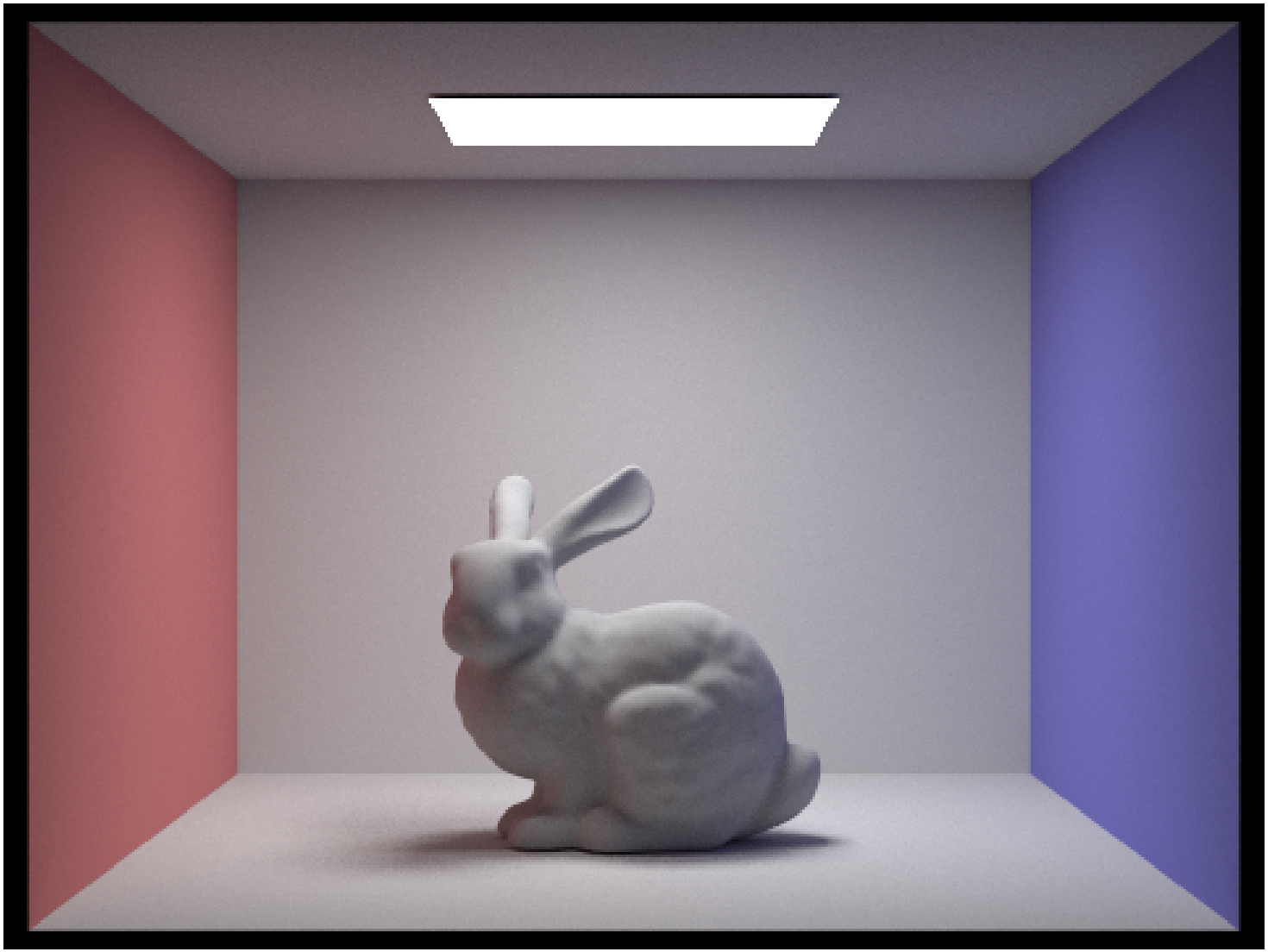

Figure 46: 1024 samples per pixel, 1 samples per area light, max ray depth 3, `isAccumBounces` true.

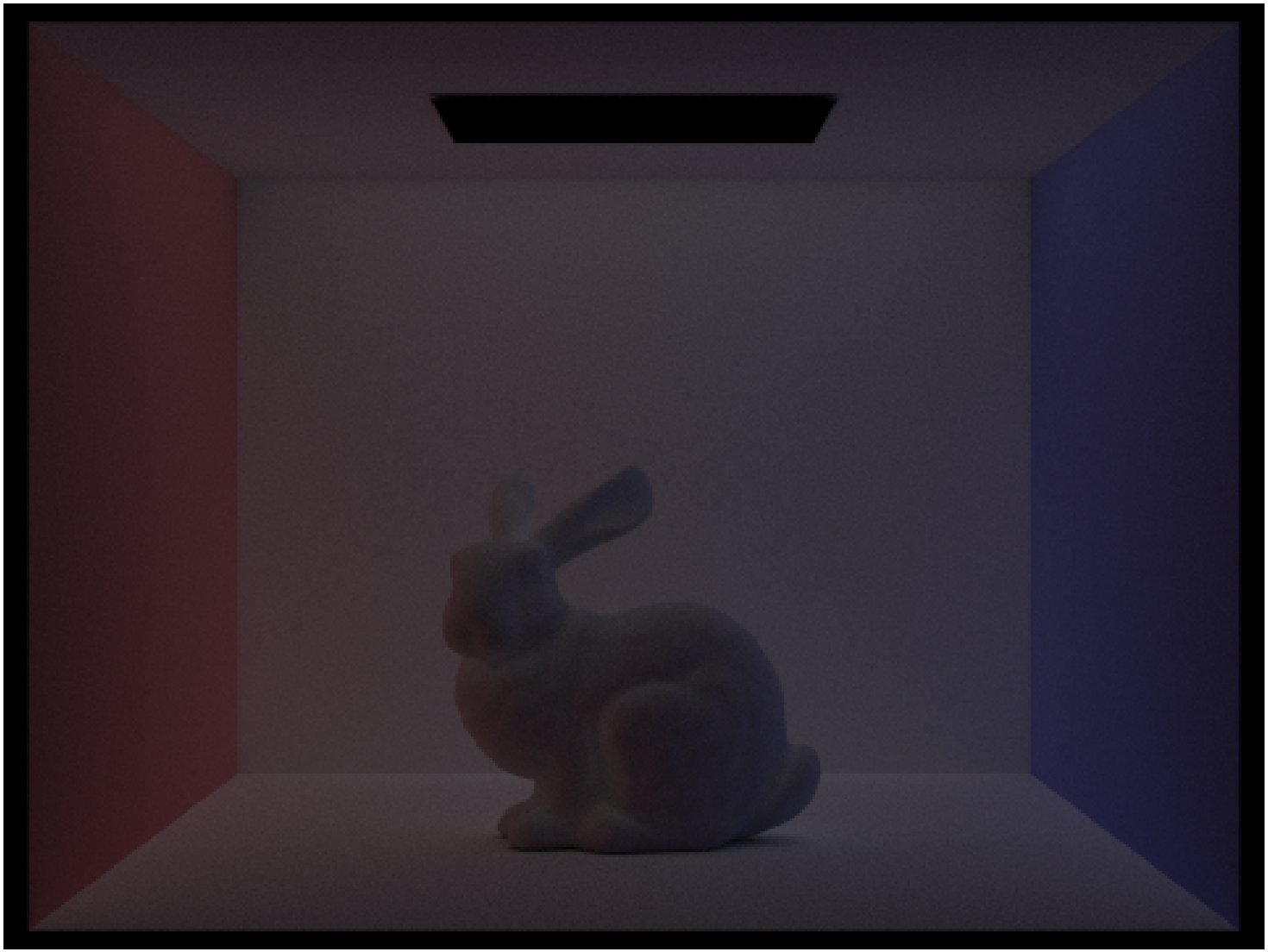

Figure 47: 1024 samples per pixel, 1 samples per area light, max ray depth 3, `isAccumBounces` false.

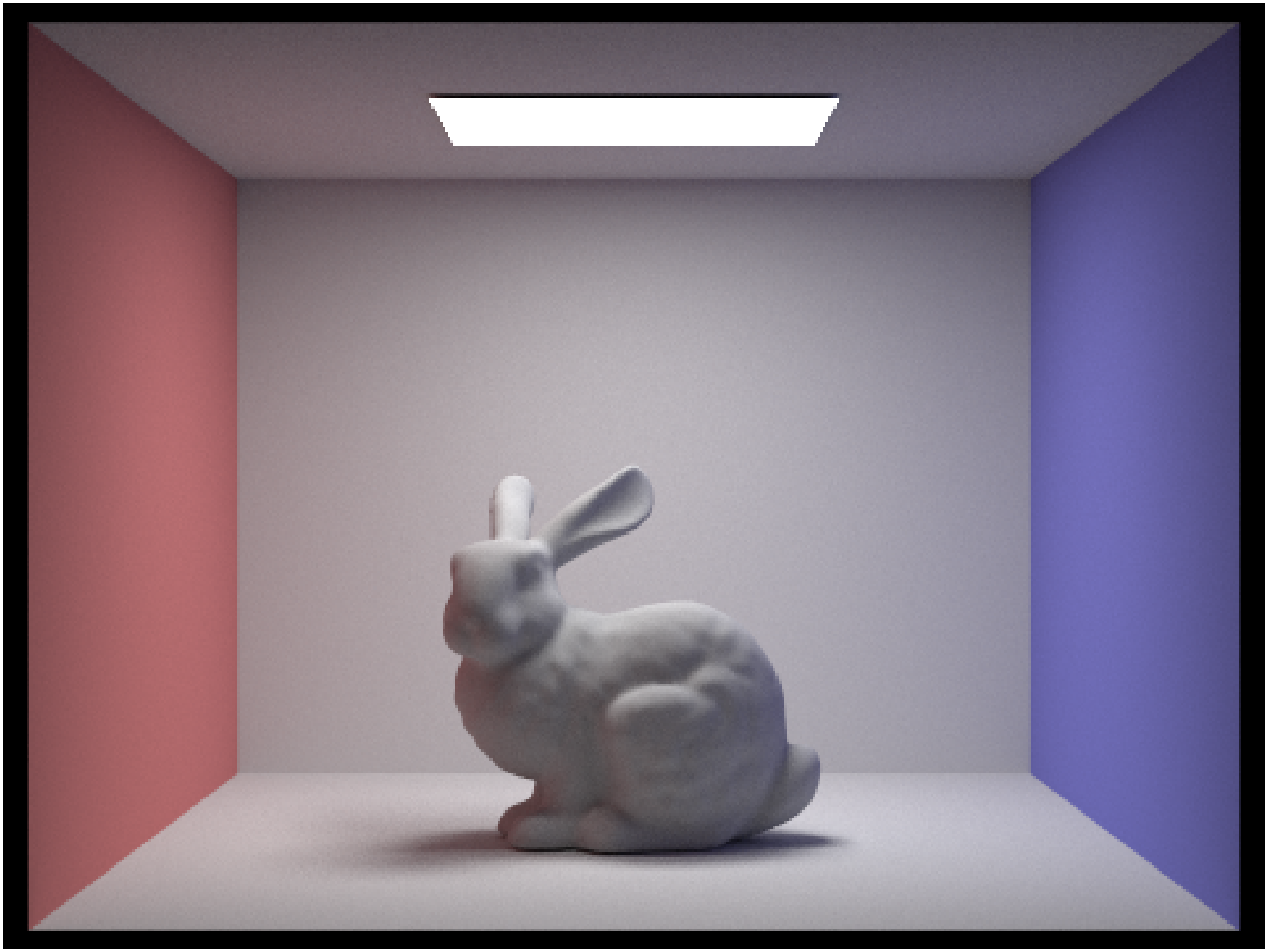

Figure 48: 1024 samples per pixel, 1 samples per area light, max ray depth 4, `isAccumBounces` true.

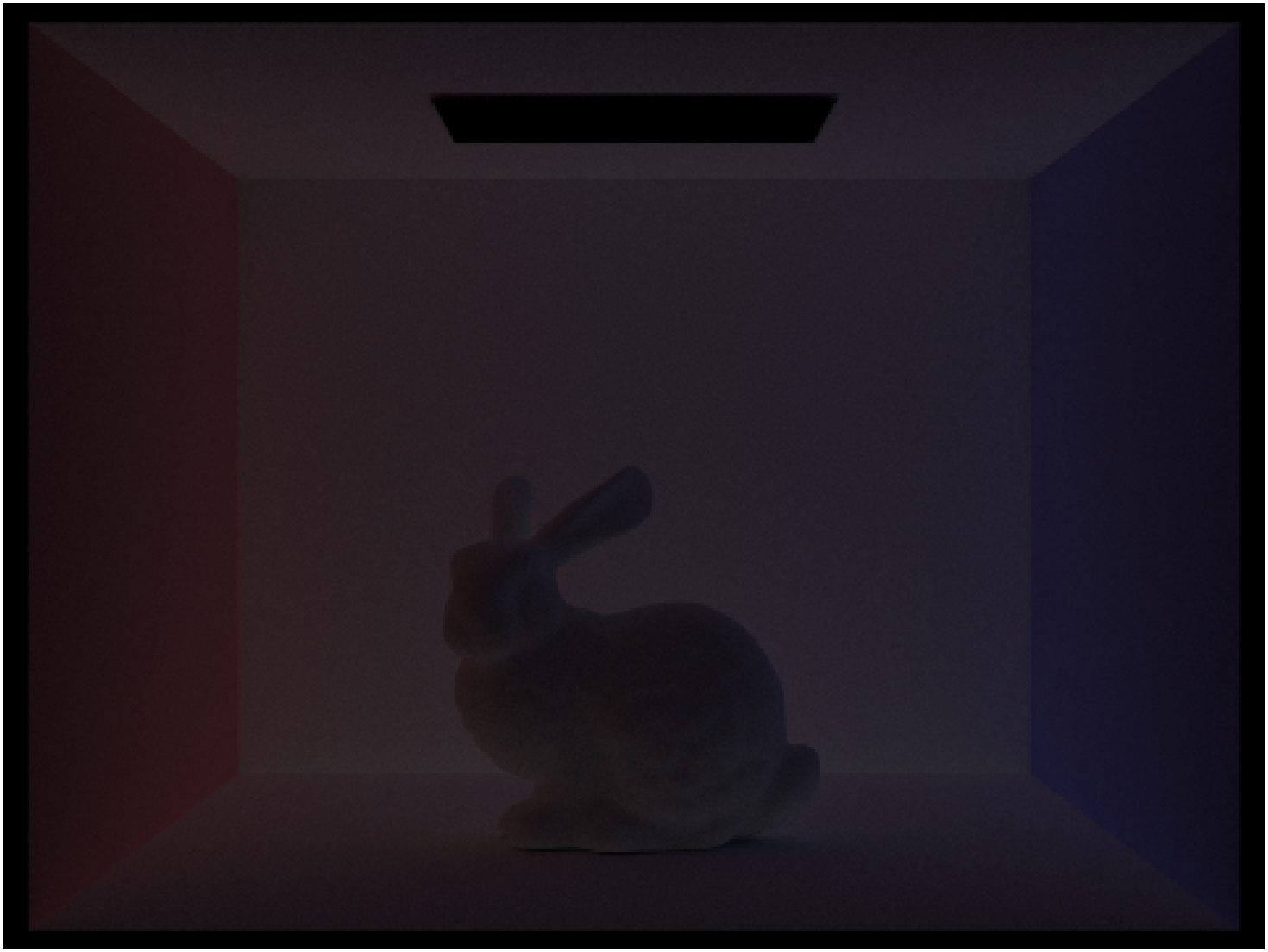

Figure 49: 1024 samples per pixel, 1 samples per area light, max ray depth 4, `isAccumBounces` false.

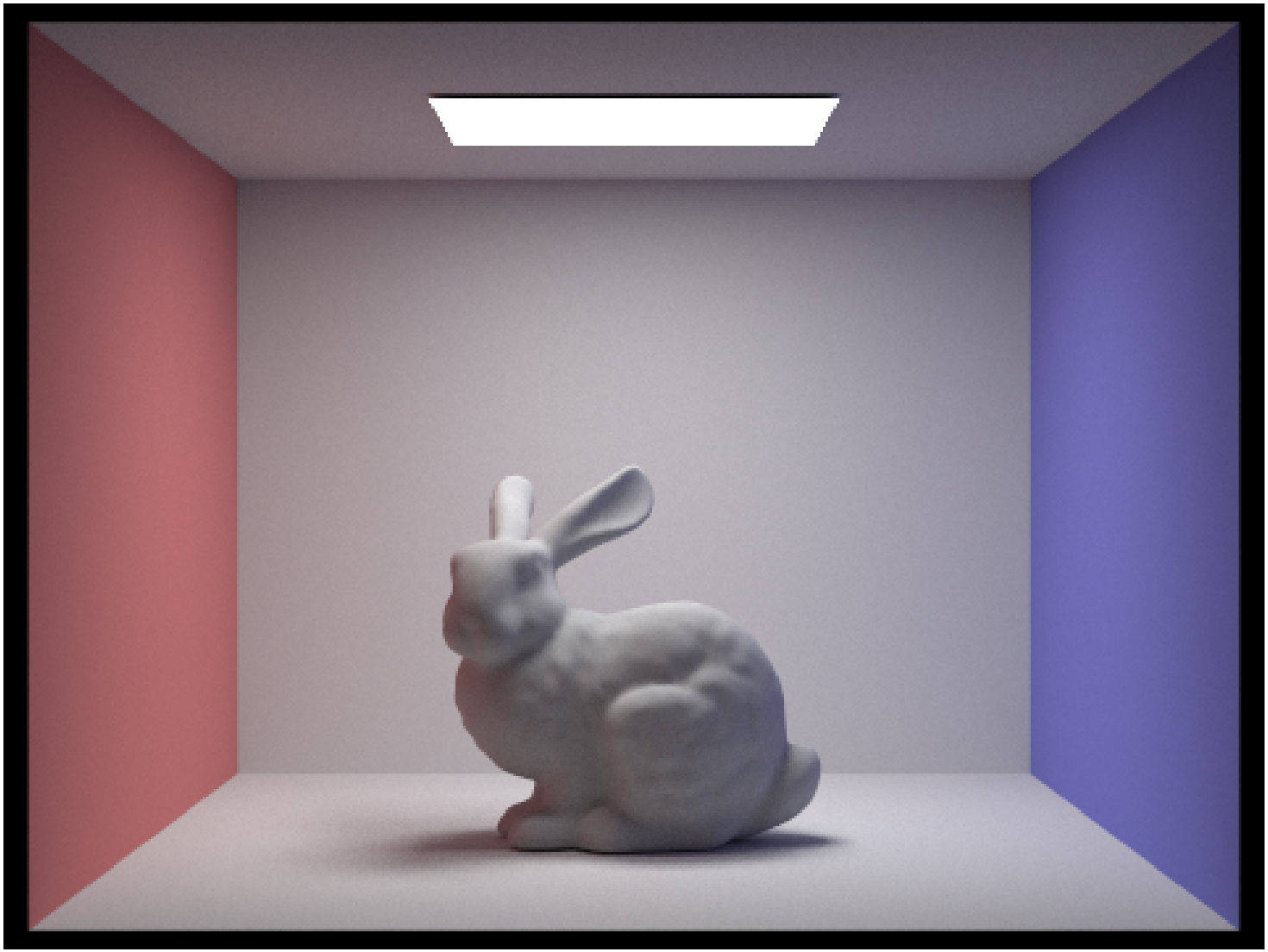

Figure 50: 1024 samples per pixel, 1 samples per area light, max ray depth 5, `isAccumBounces` true.

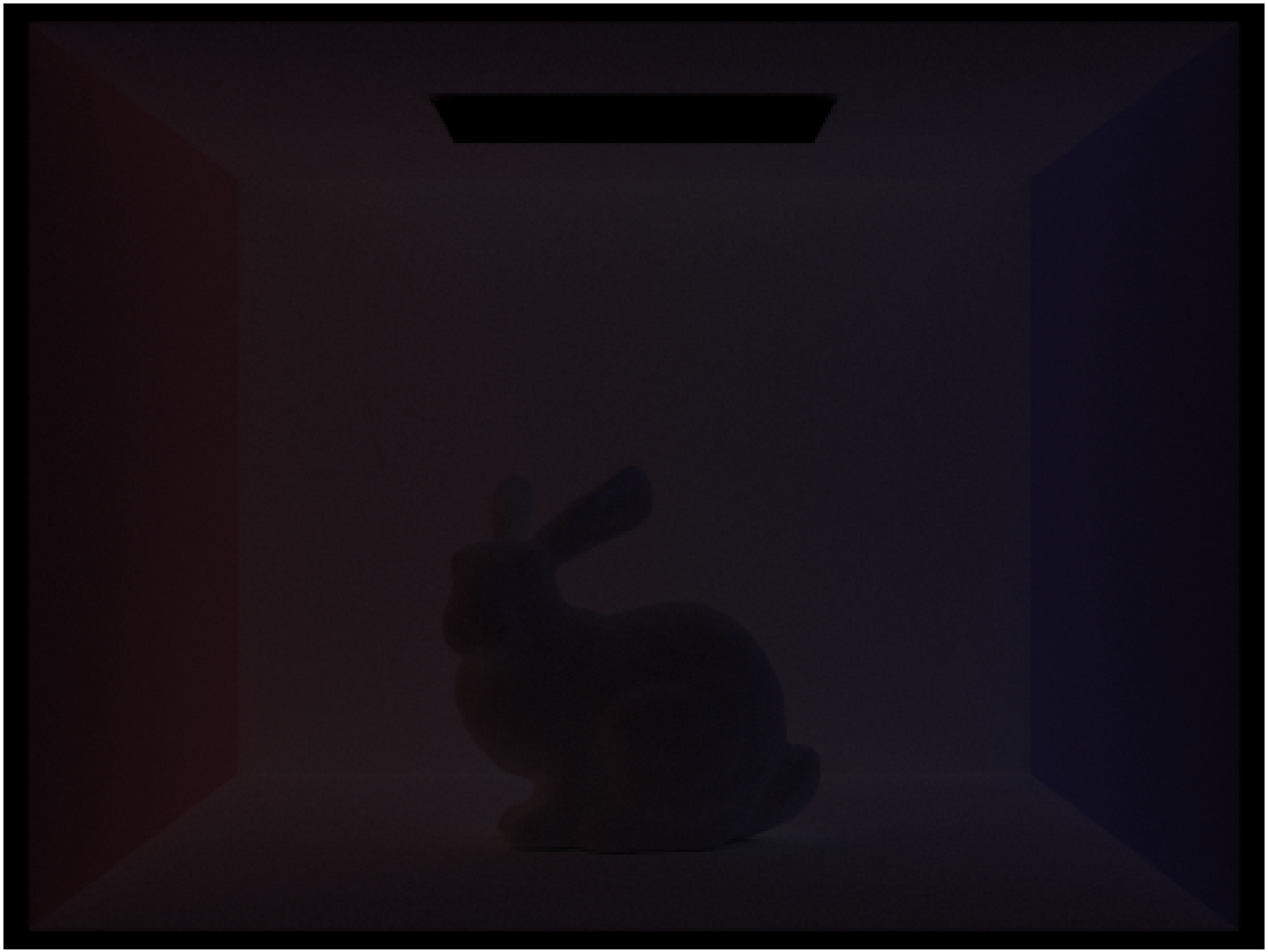

Figure 51: 1024 samples per pixel, 1 samples per area light, max ray depth 5, `isAccumBounces` false.

Below we have included pictures of a bunny inside a Cornell box with `max_ray_depth` set to 0, 1, 2, 3, 4, 5 and 100 with Russian roulette implemented with a continue probability 0.65.

As we can see, the pictures with Russian roulette start to differ from those without only after the first indirect bounce, since all the stages up to that point are guaranteed to happen in our implementation. The discrepancy between pictures become smaller, since Russian roulette tries to remove the artificial bias of in specifying the maximum ray depth.

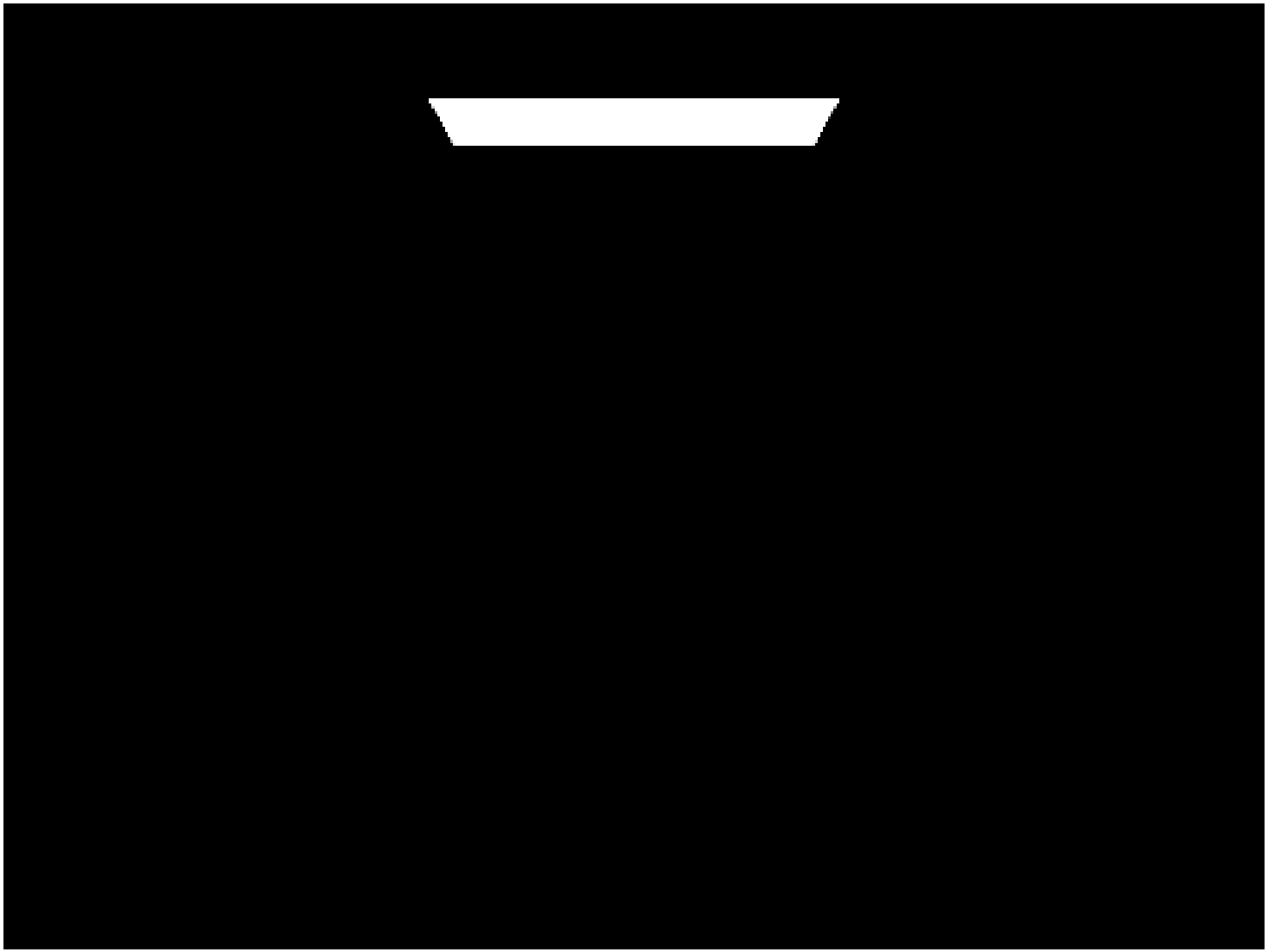

Figure 52: 1024 samples per pixel, 1 samples per area light, max ray depth 0, Russian roulette continue probability 0.65.

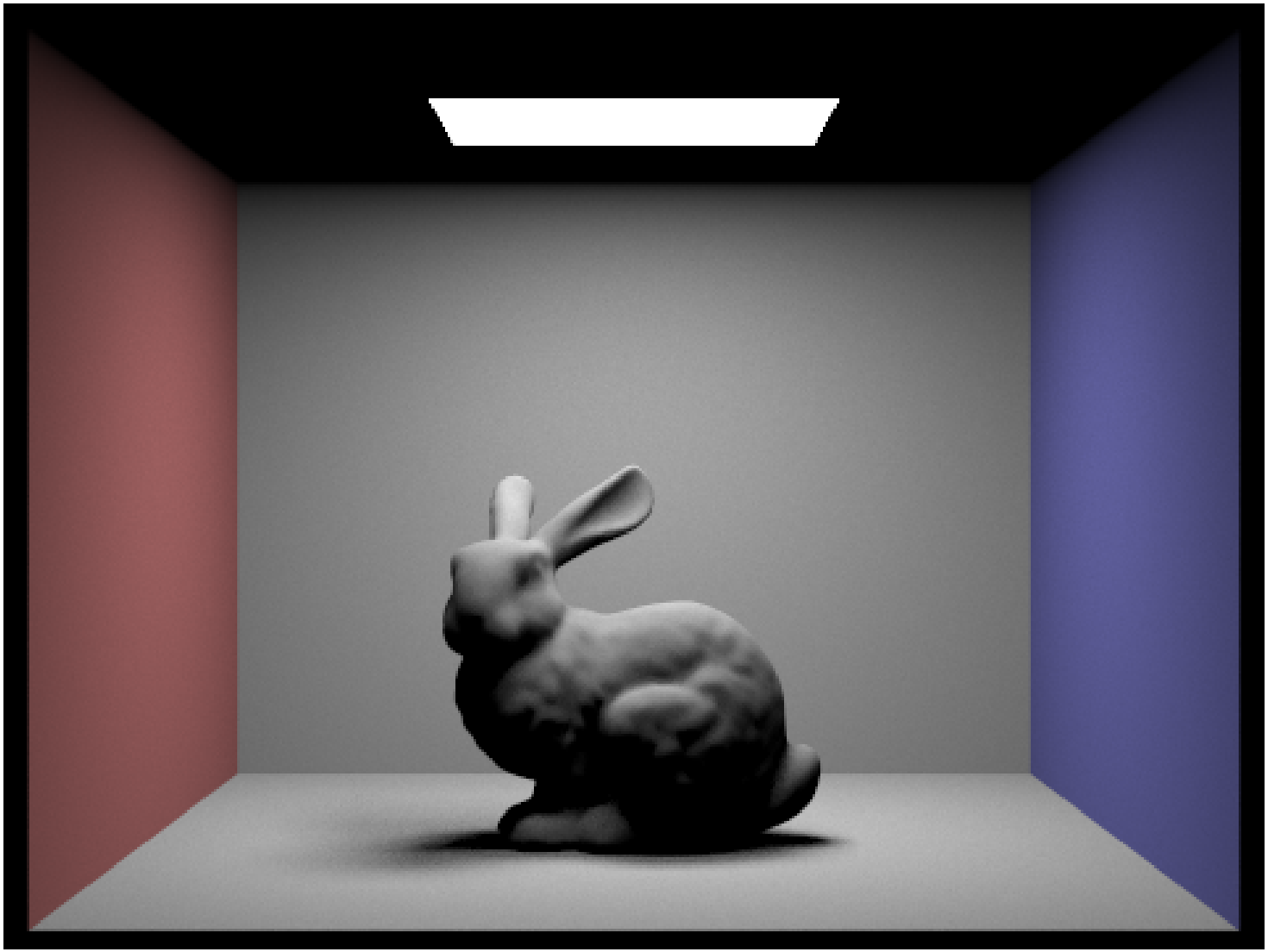

Figure 53: 1024 samples per pixel, 1 samples per area light, max ray depth 1, Russian roulette continue probability 0.65.

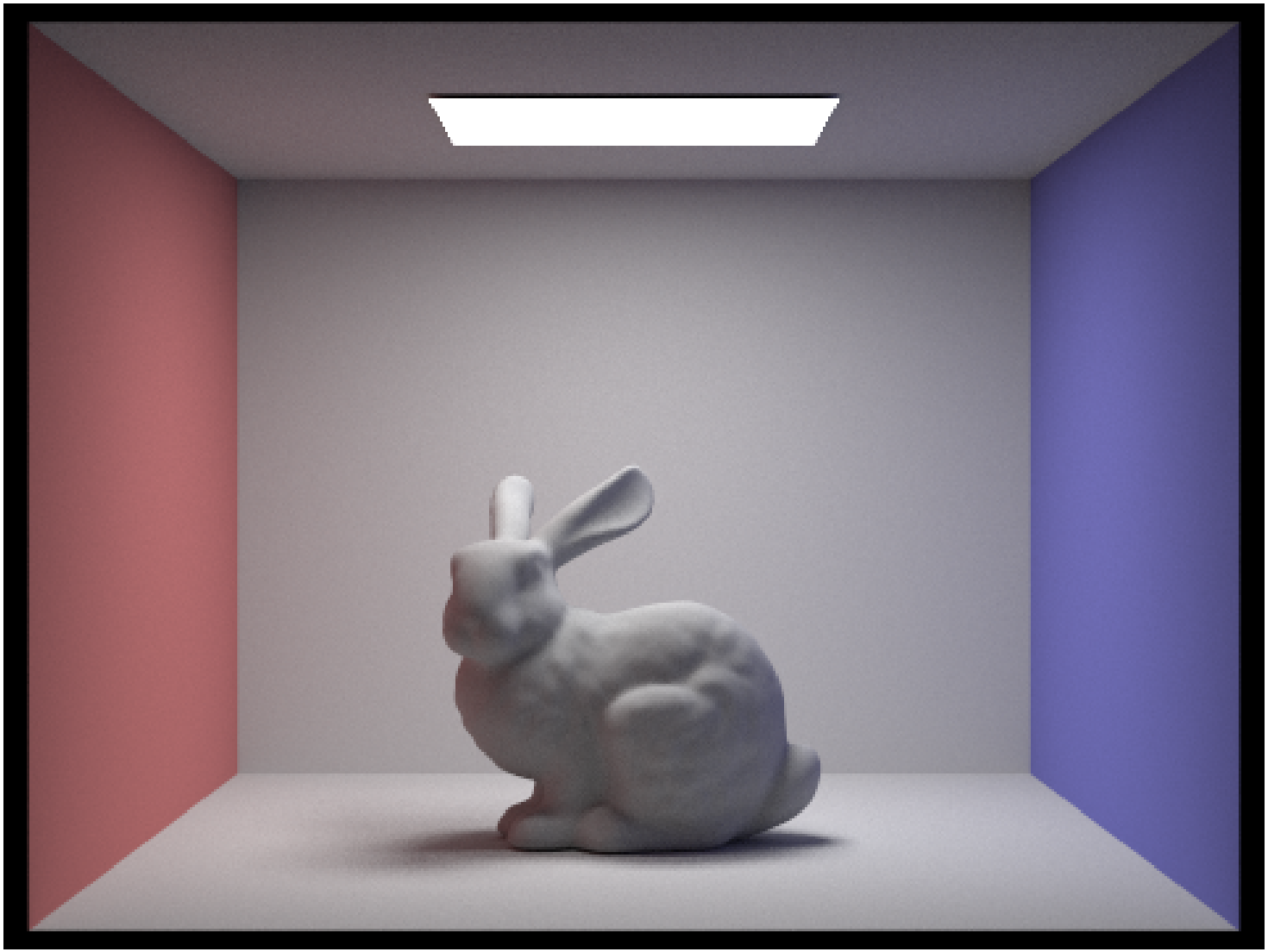

Figure 54: 1024 samples per pixel, 1 samples per area light, max ray depth 2, Russian roulette continue probability 0.65.

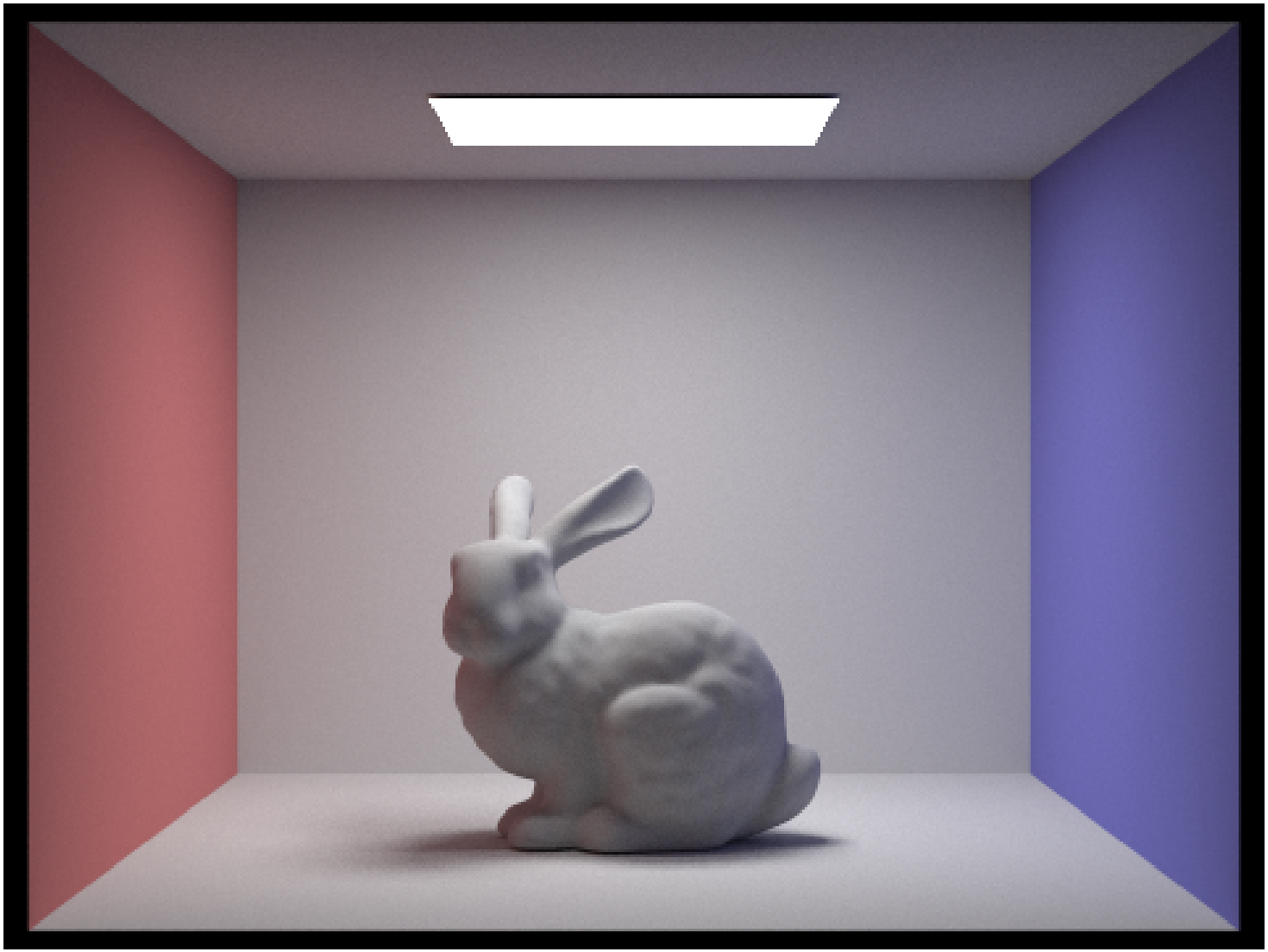

Figure 55: 1024 samples per pixel, 1 samples per area light, max ray depth 3, Russian roulette continue probability 0.65.

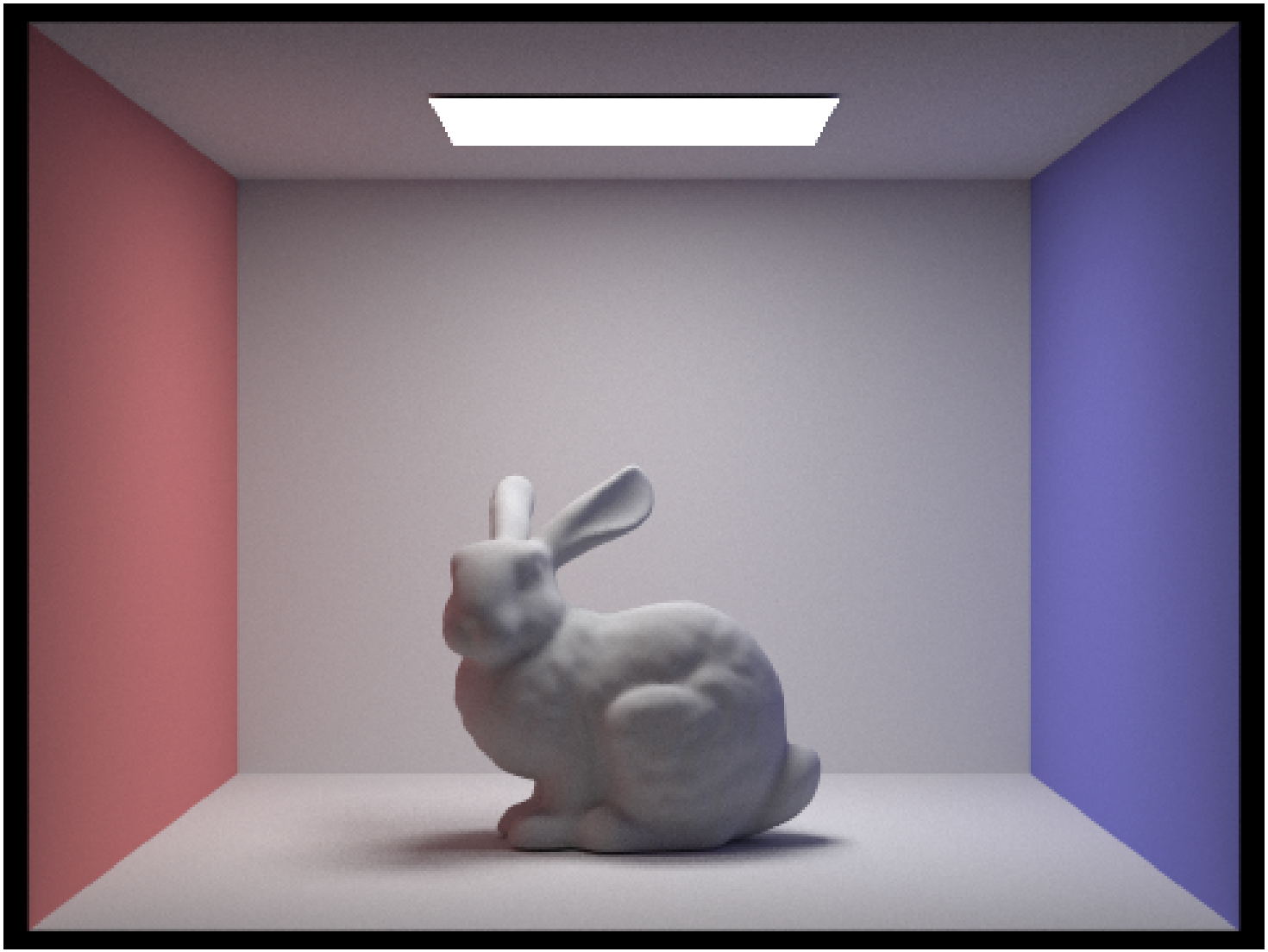

Figure 56: 1024 samples per pixel, 1 samples per area light, max ray depth 4, Russian roulette continue probability 0.65.

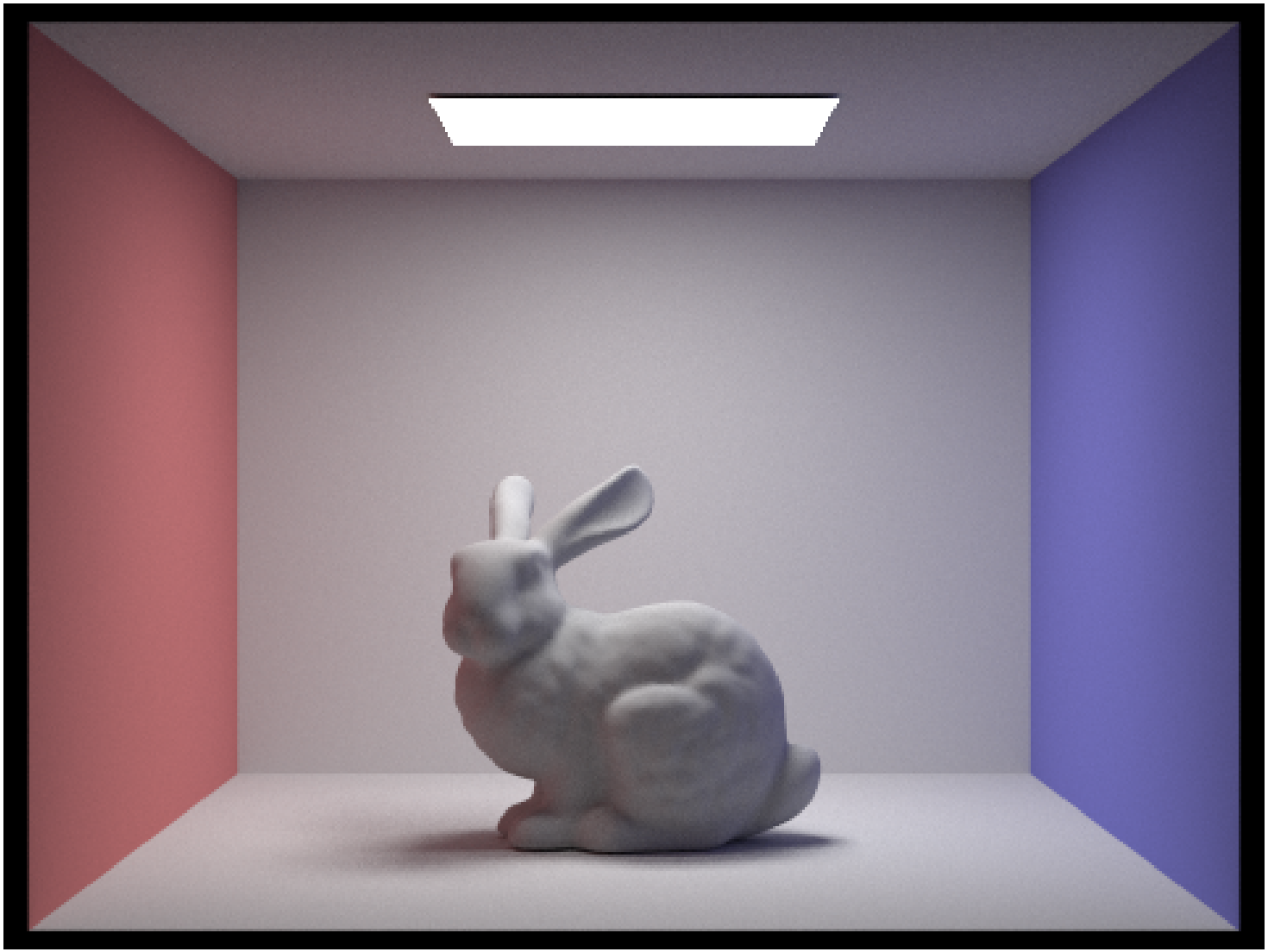

Figure 57: 1024 samples per pixel, 1 samples per area light, max ray depth 5, Russian roulette continue probability 0.65.

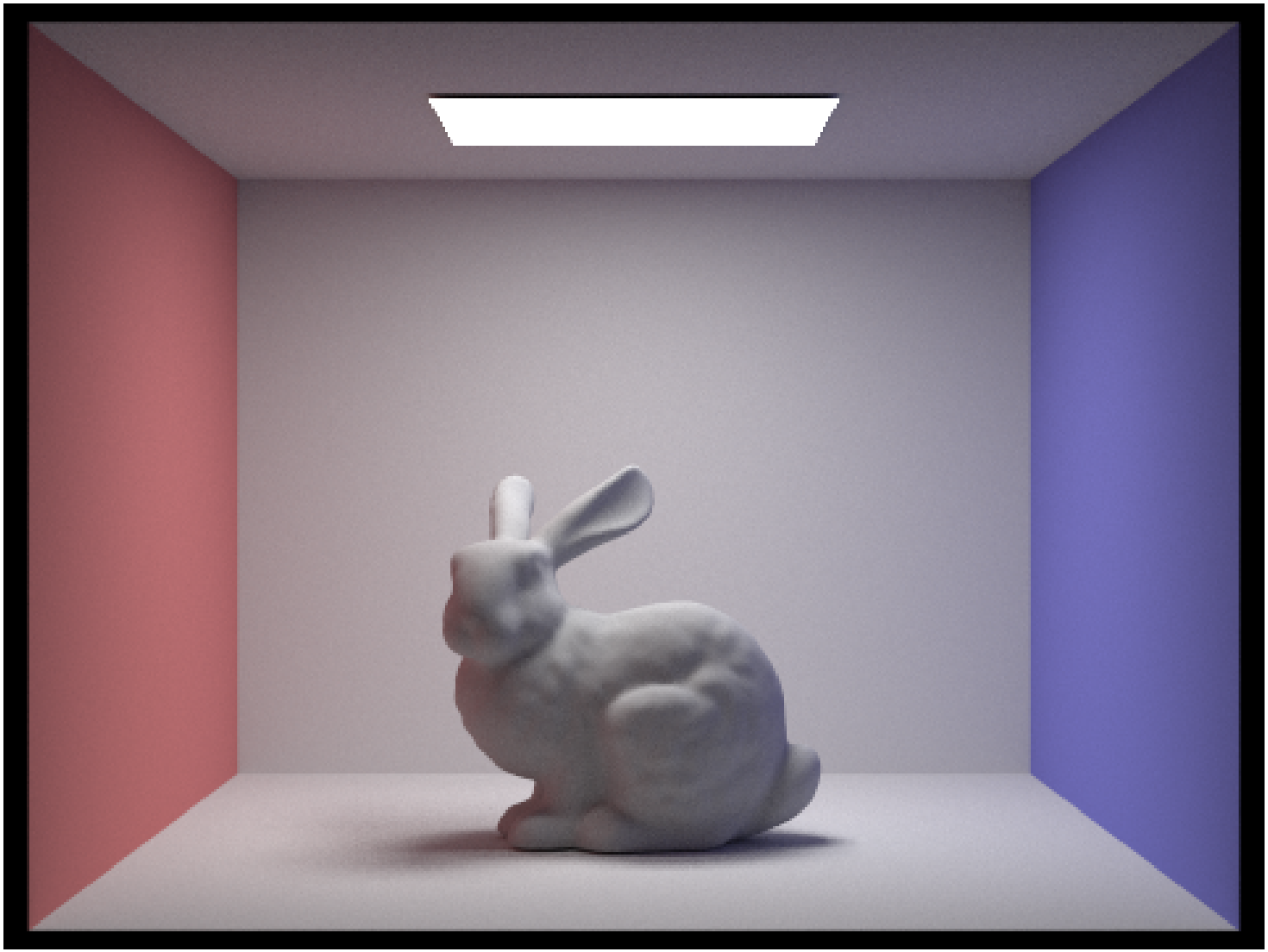

Figure 58: 1024 samples per pixel, 1 samples per area light, max ray depth 100, Russian roulette continue probability 0.65.

Below we have included pictures of the spheres inside a Cornell box with 4 light rays at 1, 2, 4, 8, 16, 32, 64, and 1024 samples per pixel.

As we can see, when the number of samples per pixel is small, the pictures are quite noisy. However, unlike the noise we discussed in direct illumination, the noises appear in these pictures are white pixels. The reason for this difference is that if some rays continue for longer than average and hit bright spots, their contribution in the Monte Carlo Integration will be large since we need to divide the estimation term every stage by the continue probability. These bright rays will skew the illuminance of certain pixels, especially if the number of samples per pixel is small. As the number of samples increase, the level of noise decreases significantly. Despite the noise level, we can still see that global illumination with Russian roulette is good at capturing the overall brightness and the general light patterns of the scene.

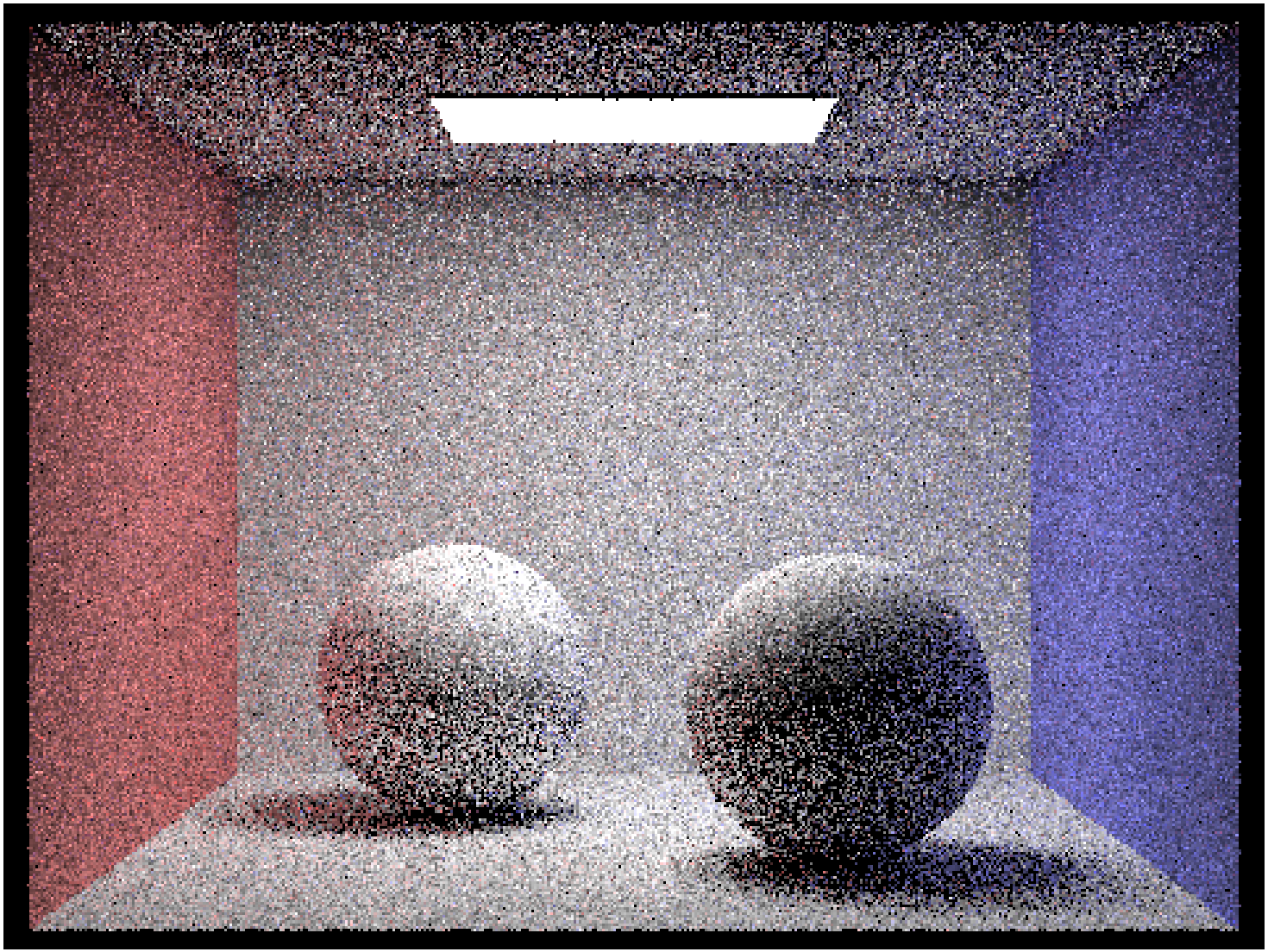

Figure 59: 1 samples per pixel, 4 samples per area light, max ray depth 100, Russian roulette continue probability 0.65.

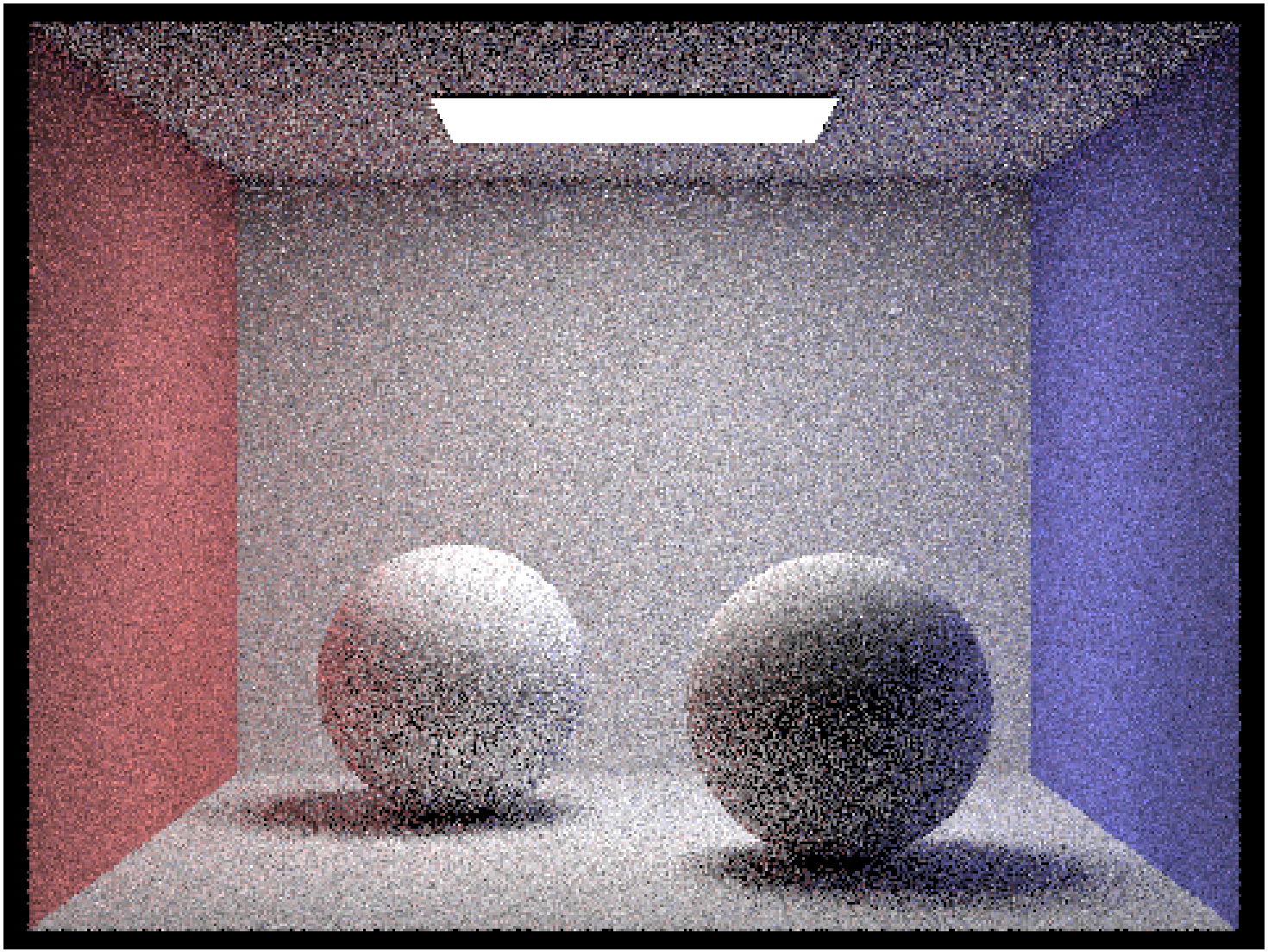

Figure 60: 2 samples per pixel, 4 samples per area light, max ray depth 100, Russian roulette continue probability 0.65.

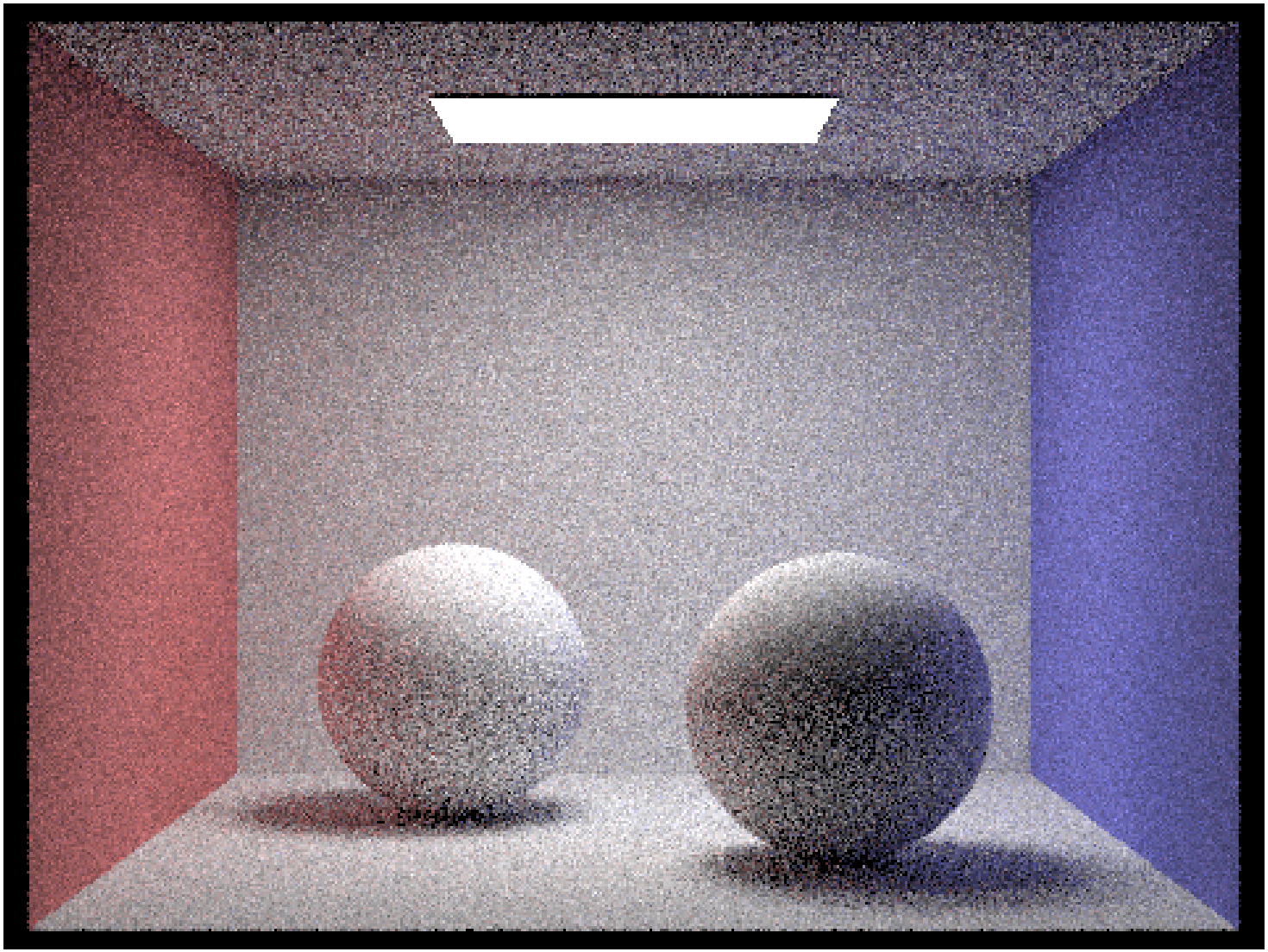

Figure 61: 4 samples per pixel, 4 samples per area light, max ray depth 100, Russian roulette continue probability 0.65.

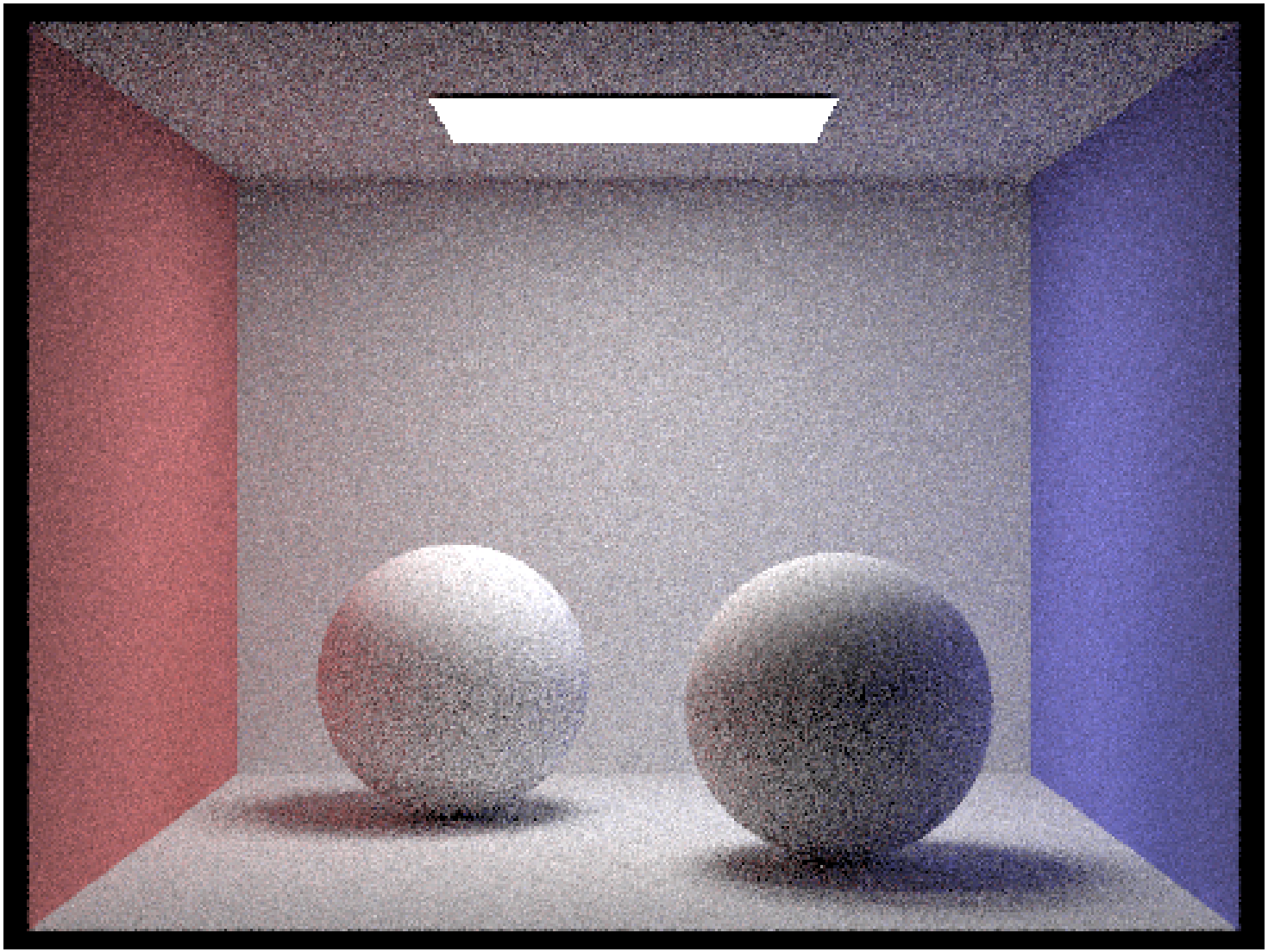

Figure 62: 8 samples per pixel, 4 samples per area light, max ray depth 100, Russian roulette continue probability 0.65.

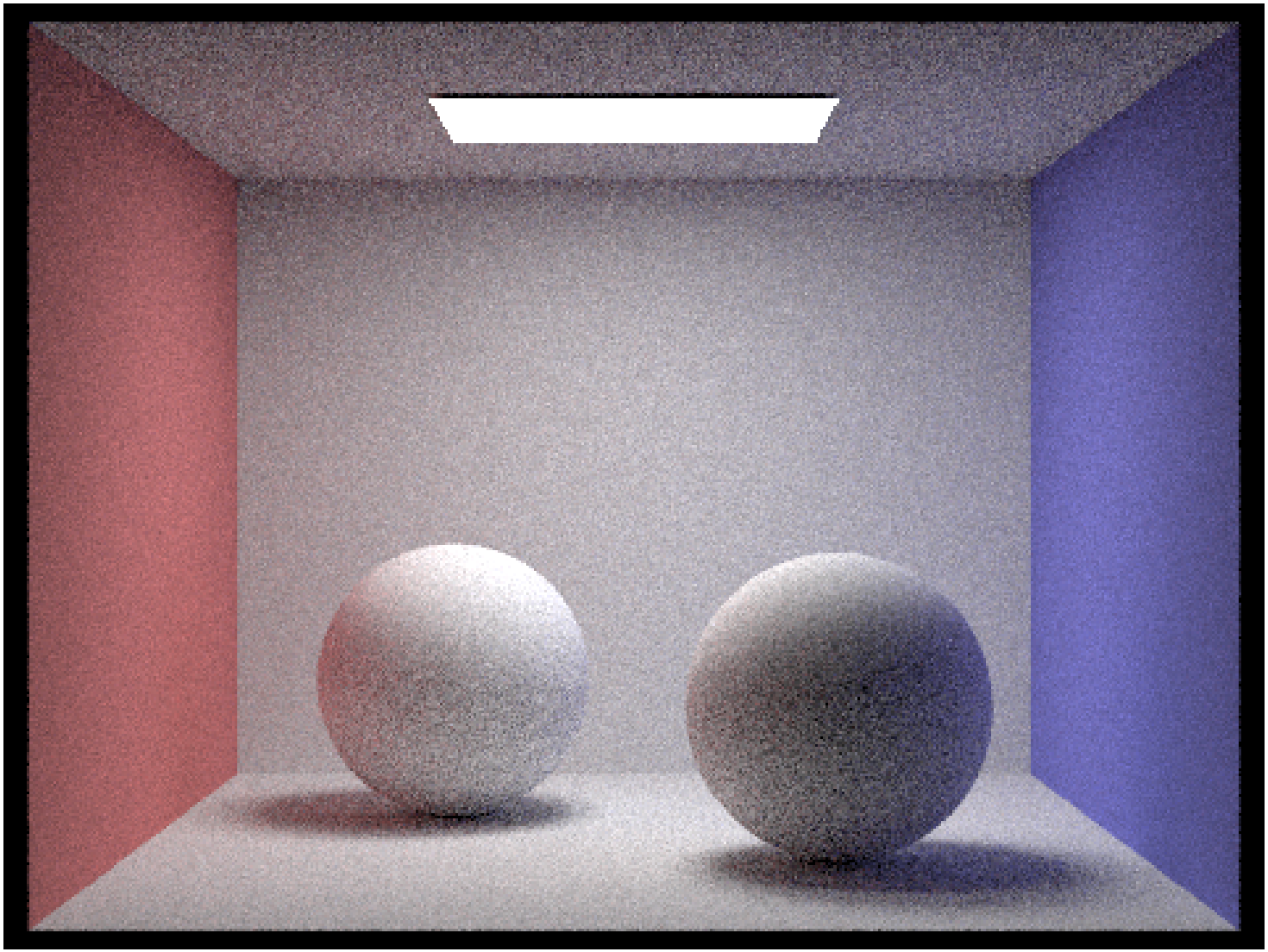

Figure 63: 16 samples per pixel, 4 samples per area light, max ray depth 100, Russian roulette continue probability 0.65.

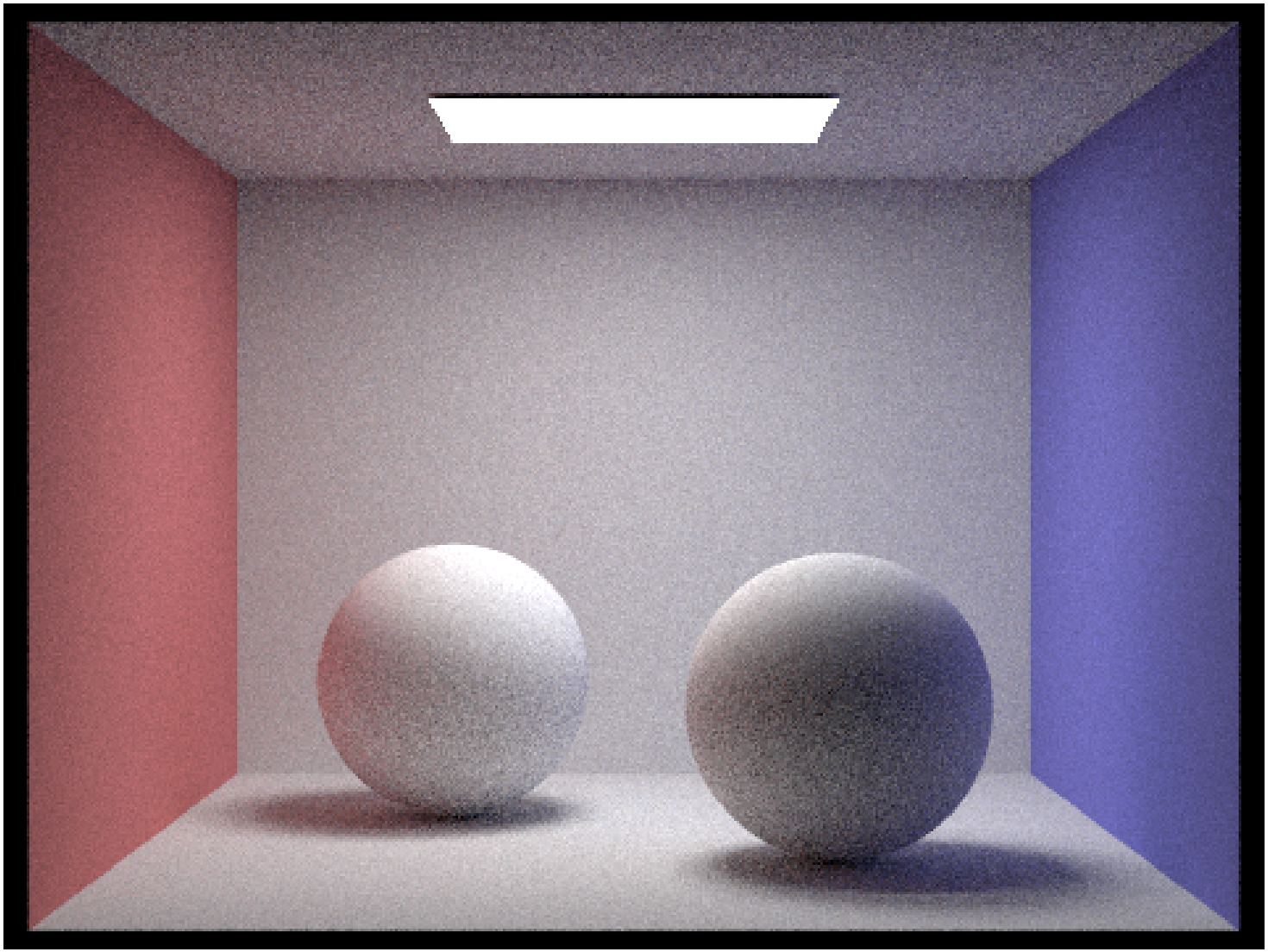

Figure 64: 32 samples per pixel, 4 samples per area light, max ray depth 100, Russian roulette continue probability 0.65.

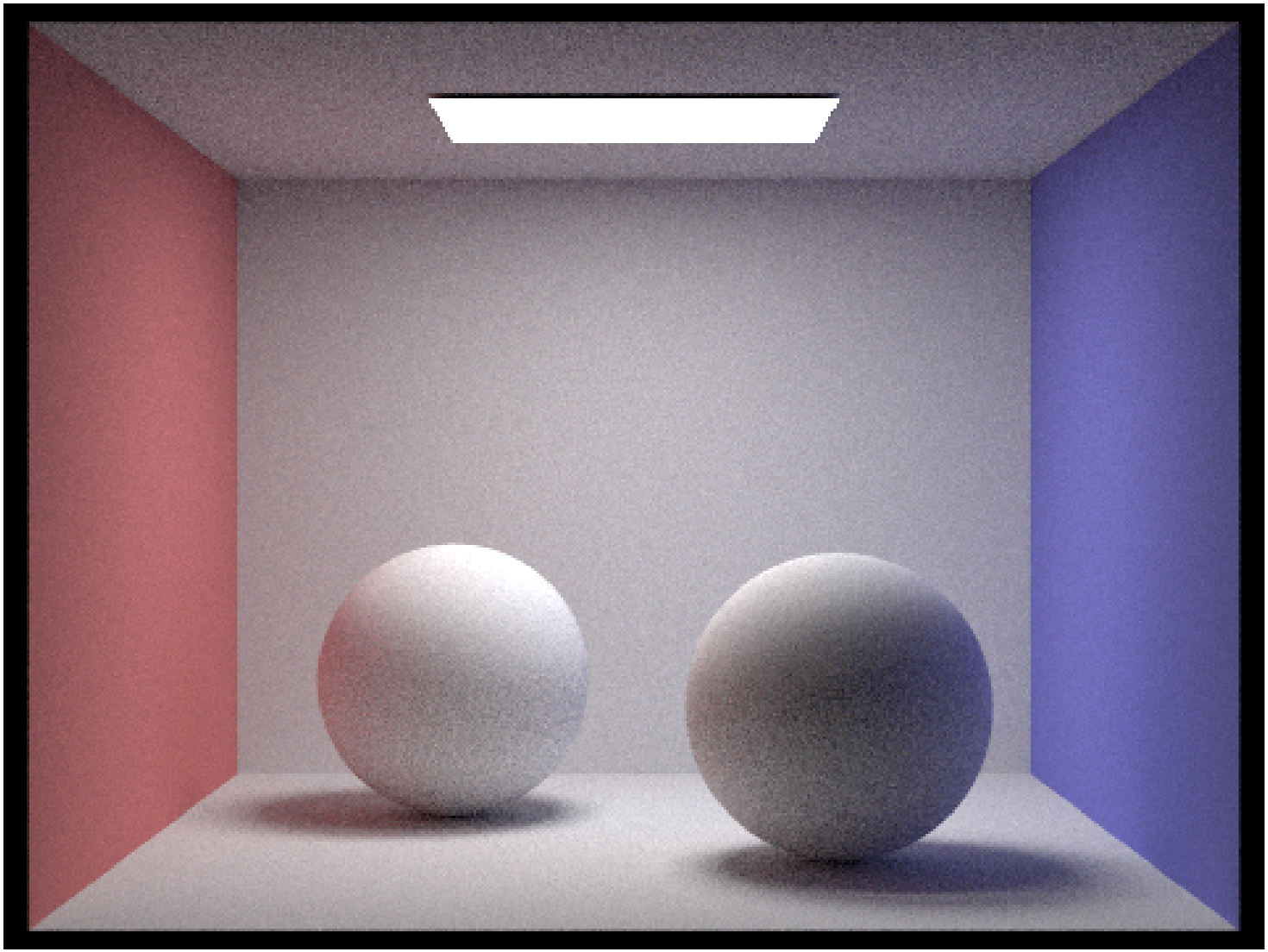

Figure 65: 64 samples per pixel, 4 samples per area light, max ray depth 100, Russian roulette continue probability 0.65.

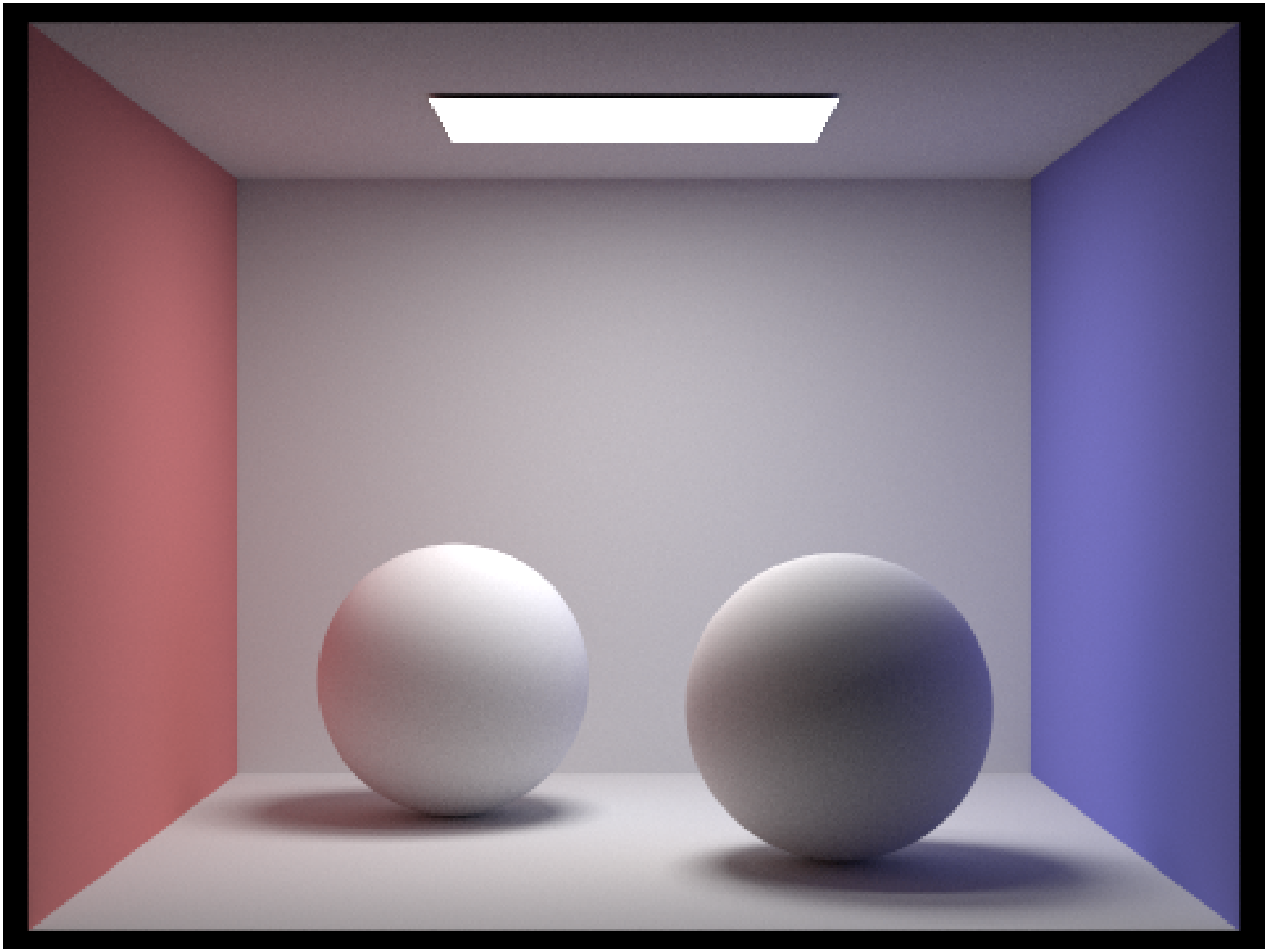

Figure 66: 1024 samples per pixel, 4 samples per area light, max ray depth 100, Russian roulette continue probability 0.65.

5 Adaptive Sampling

The direct and global illumination functions we implemented in Section 3 and Section 4 take the sampling ray as an argument, and the optimizations are based on the sampling rays. However, even with the BVH data structure and importance sampling, we can only more efficiently determine whether a ray is going to miss. The `raytrace_pixel` method in Section 1 will always send `ns_aa` sampling rays each pixel. In practice, different pixels will have different levels of variances of the radiance of the sampling rays sent. If the variance of the samples is small for a pixel compared to the number of sampling rays sent, it is reasonable to terminate the sampling process early and allocate computing resources to the pixels with large sampling variances.

In order to quantitatively determine whether it is appropriate to terminate the sampling process early for a pixel, we introduce the concept of statistical test and confidence interval. Specifically, we want to use the Central Limit Theorem to estimate the confidence interval with the null hypothesis H_0: the current state is not yet stable and the alternative hypothesis H_a: the current state is stable.

For every sampling ray, we estimate the global radiance `Li` with the function `est_radiance_global_illumination`. However, the radiance `Li` is a 3D vector to which it is hard to apply the z-test. Instead, we will compute the illuminance with `Li.illum()` and use this scalar to determine whether the state is stable. In our `raytrace_pixel` method, we let `s1` be the sum of the illuminance, and `s2` be the sum of the square illuminance. For every `samplePerBatch` samples taken and a total number of `i` samples, we compute the mean `mu = s1 / i`, the standard deviation `sigma = std::sqrt((s2 - s1 * s1 / i) / (i - 1))` and the test value `test = 1.96 * sigma / std::sqrt(i)`. If the test value `test <= maxTolerance * mu`, then we can reject the null hypothesis with the confidence `1 - maxTolerance` and accept the alternative hypothesis that the sampling state is already stable. We record the number of total samples taken `i` in the `SampleCountBuffer` vector.

Below we have included two scenes with adaptive sampling with their sampling rate image.

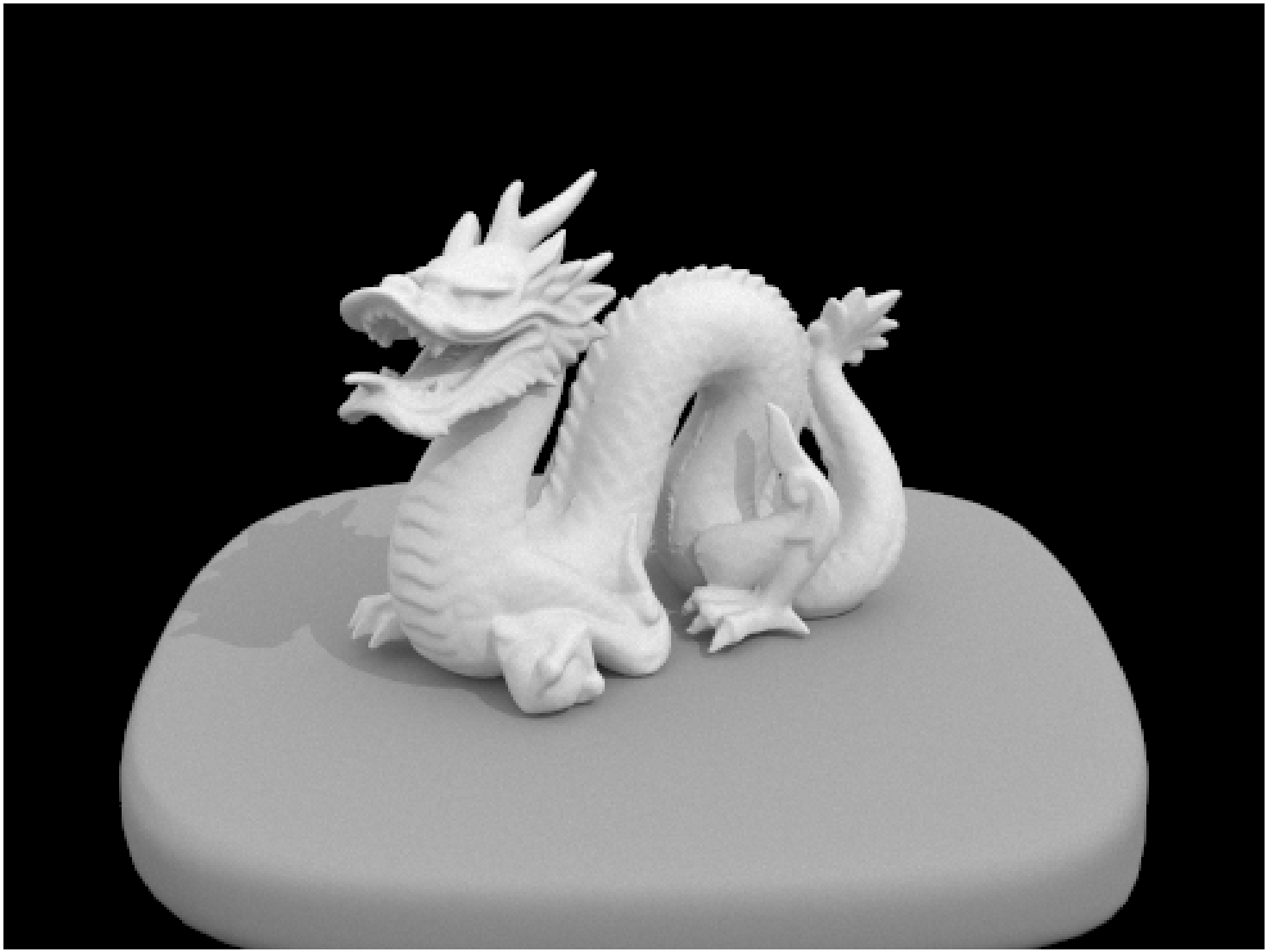

Figure 67: 2048 samples per pixel, 1 sample per area light, 64 samples per batch, max tolerance 0.05, max ray depth 100, Russian roulette continue probability 0.65.

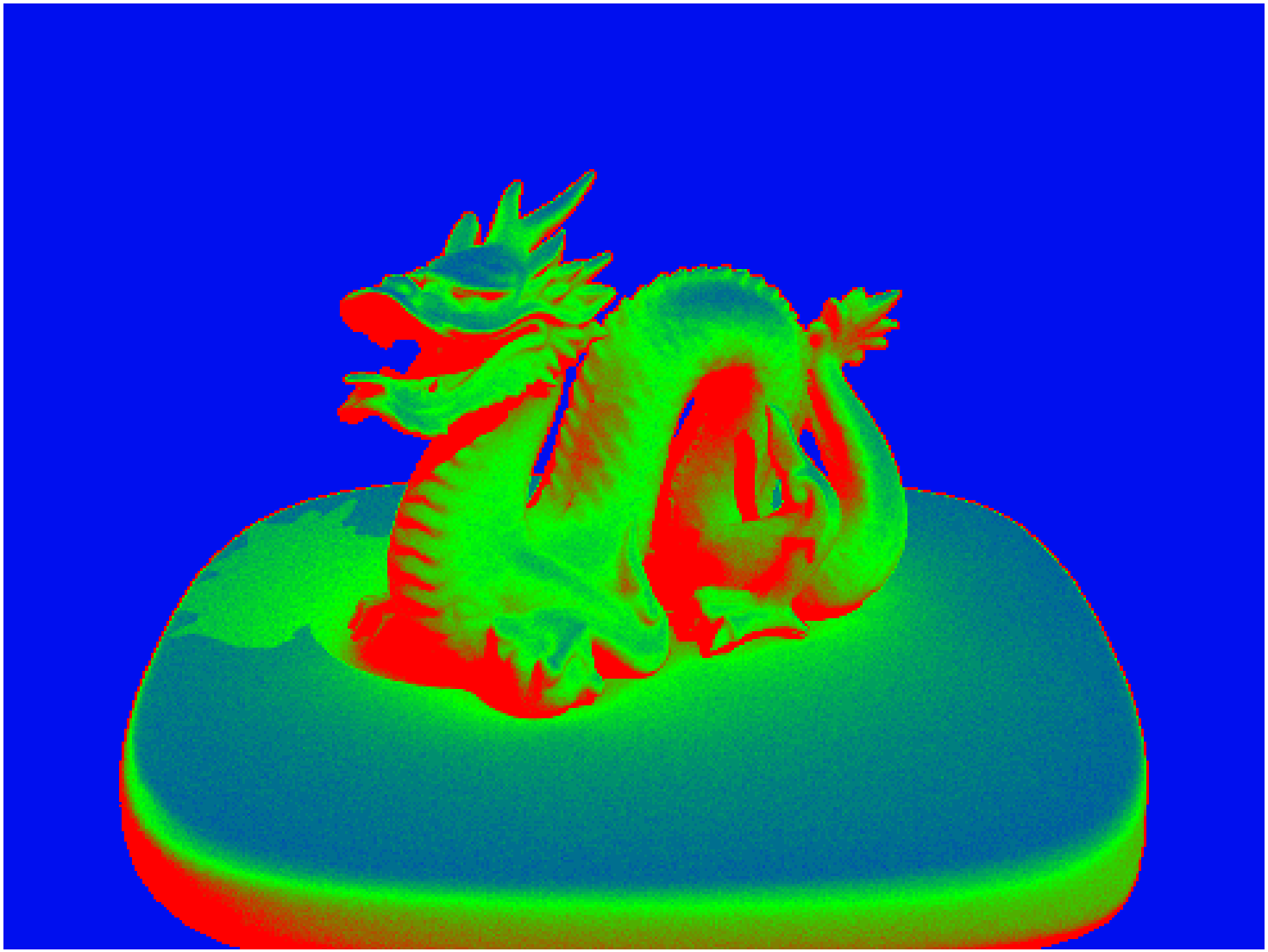

Figure 68: Sampling rate image.

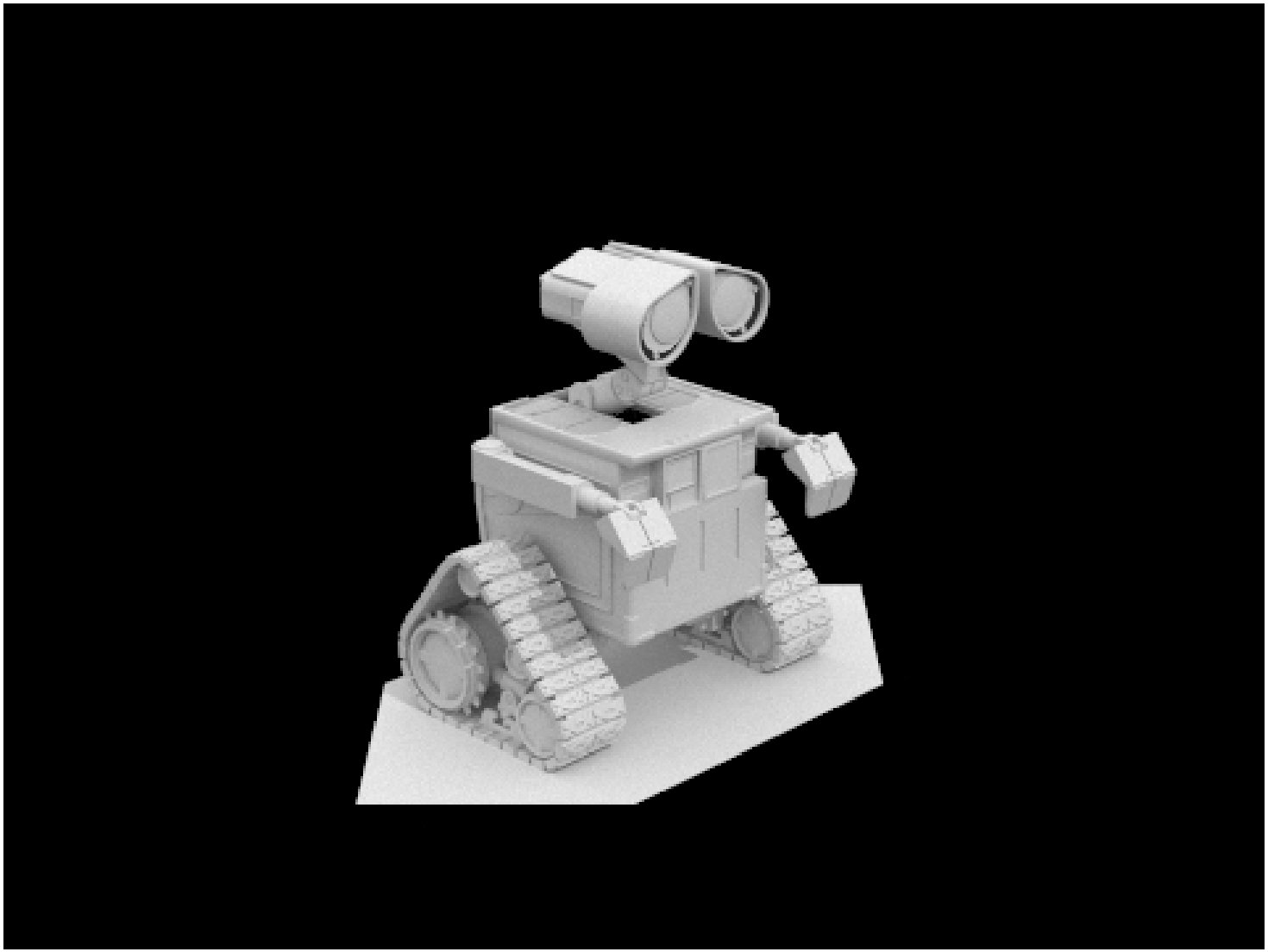

Figure 69: 2048 samples per pixel, 1 sample per area light, 64 samples per batch, max tolerance 0.05, max ray depth 100, Russian roulette continue probability 0.65.

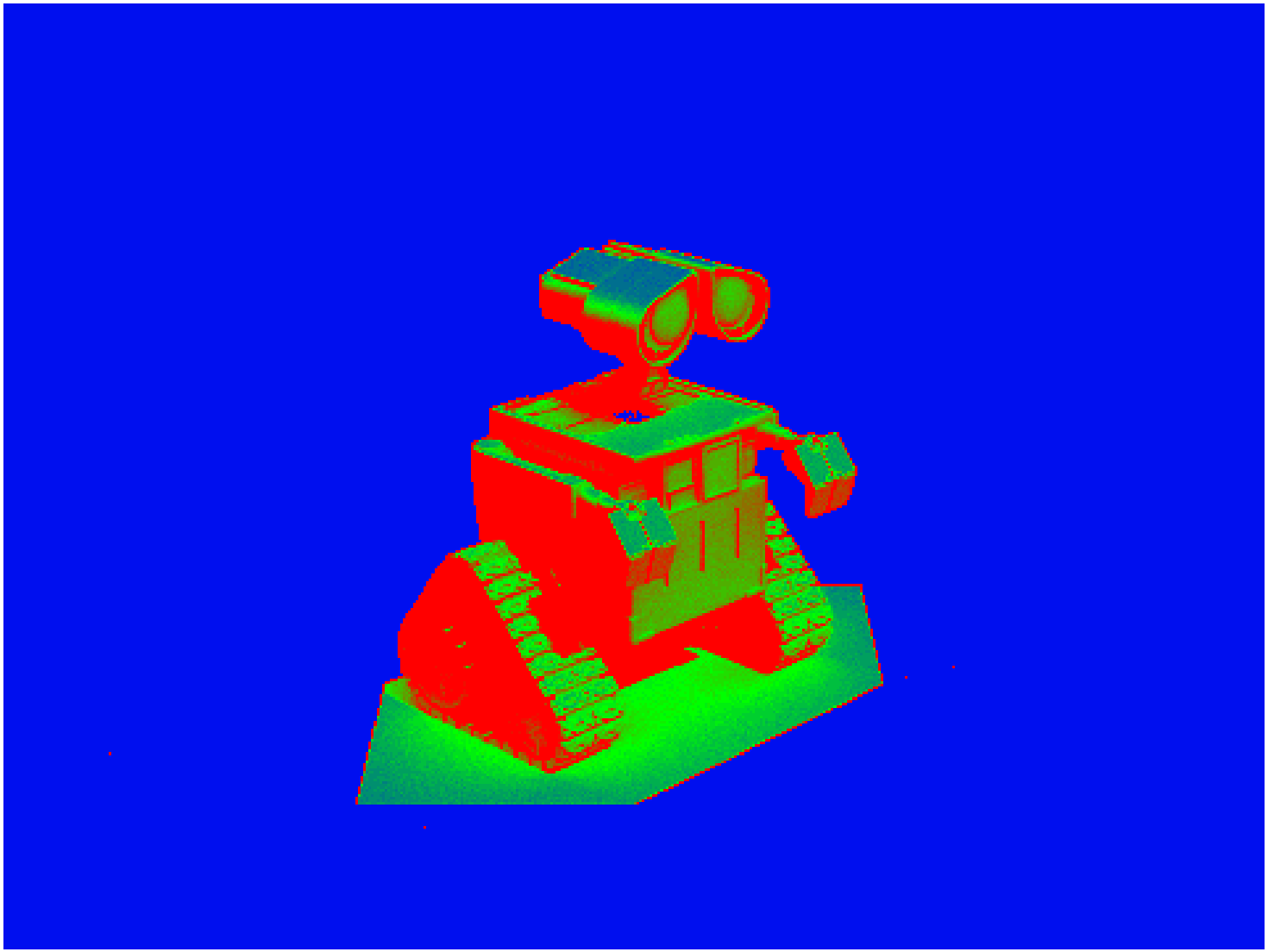

Figure 70: Sampling rate image.