Ray Tracing using HIP

ATSUSHI YOSHIMURA, Advanced Micro Devices, Inc., Japan
KENTA ETO, Advanced Micro Devices, Inc., Japan
DANIEL MEISTER, Advanced Micro Devices, Inc., Japan
TAKAHIRO HARADA, Advanced Micro Devices, Inc., USA

1 INTRODUCTION

Ray tracing [1] is an essential and widely used technique in computer graphics. There are many applications using ray tracing, and one of the most popular and exciting use cases is photorealistic rendering. Please, take a look at Figure 1. Which picture do you think is a photograph? One of them is rendered by ray tracing (left), and the other is a photograph taken by a physical camera (right). The rendered image is practically indistinguishable from the photograph. This demonstrates how powerful ray tracing is. Light phenomena are described by the physical laws of optics, which are very complex in general. In computer graphics, we use a simplified model described by geometrical optics, modeling light propagation in terms of rays (i.e., straight lines). Even with this simplified model, we are able to render numerous lighting effects such as soft shadows, color bleeding, reflection/refraction, and depth-of-field (Figure 2) in a single unified framework.

In this technical report, we introduce the basics of ray tracing and explain how to accelerate the computation of the rendering algorithm in HIP. We also show how to use a HIP ray tracing framework - HIPRT, leveraging hardware ray tracing features of AMD GPUs. We conclude this technical report with a list of references for further reading.

2 OROCHI

The natural way to implement a HIP application accelerated by the GPU is to use the HIP SDK or ROCm. A drawback of using vanilla HIP is that the host code is compiled either for AMD or NVIDIA backends, requiring recompilation each time we switch to a GPU of the other vendor. In this section, we introduce Orochi, an open-source library that allows to switch backends of different vendors at runtime [2]. Orochi is a wrapper built on top of the HIP API, loading HIP API functions directly from the driver libraries (`dll` files on Windows or `so` files on Linux), and thus it does not require the HIP SDK installed. Orochi API is designed such that developers who are familiar with HIP or CUDA APIs can use it easily. We use `hip` prefix for the HIP API while Orochi uses `oro` prefix for the Orochi API. What it does is simply redirect the function to the appropriate API underneath. Listing 1 shows a sample code, initializing a GPU device and Orochi context. Listing 2 shows how this code is transformed to the Orochi API.

email: { Atsushi.Yoshimura, Kenta.Eto, Daniel.Meister, Takahiro.Harada }@amd.com
Advanced Micro Devices, Inc. Technical Report No. 26-01-4793. v.1, January 10, 2026.

Fig. 1. Ray tracing or photo?

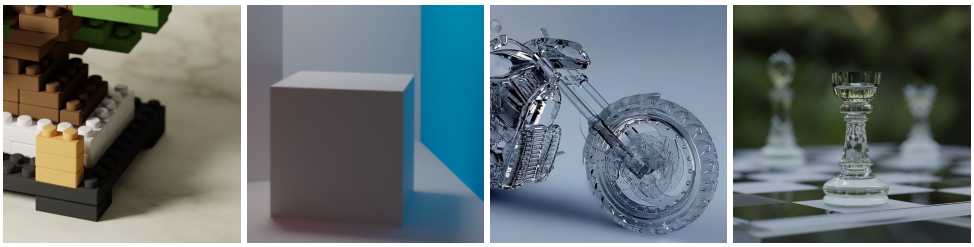


Fig. 2. Examples of lighting effects that can be observed in the real world, produced by a ray-tracing-based renderer: soft shadows (top-left), color bleeding (top-right), reflection/refraction (bottom-left), and depth-of-field (bottom-right).

Listing 1. A HIP sample code.

```
1 #include <hip/hip_runtime.h>
2 hipInit(0);
3 hipDevice device;
4 hipDeviceGet(&device, 0);
5 hipCtx ctx;
6 hipCtxCreate(&ctx, 0, device);
```

Listing 2. An Orochi sample code.

```
1 #include <Orochi/Orochi.h>
2 oroInitialize(ORO_API_HIP, 0);
3 oroInit(0);
4 oroDevice device;
5 oroDeviceGet(&device, 0);
6 oroCtx ctx;
7 oroCtxCreate(&ctx, 0, device);
```

Most of the code in the above example is the same except for the API prefix. Only addition to the code is line 2 in Listing 2 of the program where we call `oroInitialize()`. We pass `ORO_API_HIP` to the function, which tells the library to load the HIP API from the driver libraries. If we want to use an NVIDIA GPU, we need to pass `ORO_API_CUDA` instead. We can change the code path from HIP to CUDA without recompiling the application, allowing for dynamic switching between AMD GPU and NVIDIA GPU in runtime. We can also use both AMD and NVIDIA GPUs by initializing Orochi with both keys (`ORO_API_HIP | ORO_API_CUDA`). If the system has two GPUs, one AMD

and one NVIDIA, Orochi loads both APIs, allowing us to use both devices at the same time in a single application. Listing 3 shows a sample code that loads the HIP and CUDA APIs on the first line, counts the number of AMD and NVIDIA devices, and prints the name and architecture of each GPU. The full source code of this and other examples can be found in the Orochi repository on GPUOpen[2].

Listing 3. Querying device count and printing a device name and architecture of each device.

```
oroInitialize(static_cast<oroApi>(ORO_API_CUDA | ORO_API_HIP), 0);
oroInit(0);
int nDevicesTotal;
oroGetDeviceCount(&nDevicesTotal);
int nAMDDevices;
oroGetDeviceCount(&nAMDDevices, ORO_API_HIP);
int nNVIDIADevices;
oroGetDeviceCount(&nNVIDIADevices, ORO_API_CUDA);
std::cout << "# of devices: " << nDevicesTotal << std::endl;
std::cout << "# of AMD devices: " << nAMDDevices << std::endl;
std::cout << "# of NV devices: " << nNVIDIADevices << std::endl;
for(int i = 0; i < nDevicesTotal; i++)
{
    oroDevice device;
    oroDeviceGet(&device, i);
    oroDeviceProp props;
    oroGetDeviceProperties(&props, device);
    std::cout << "Device name and architecture: " << props.name << " (" << props.
    gcnArchName << ")" << std::endl;
}
```

3 RENDERING TRIANGLES

We use rays as a straight line to mimic physical light behavior in the real world. As light bounces around multiple times until they absorbed in the scene's surface, we have to cast a lot of rays for the light simulation. Also, triangles are commonly used primitives to represent the geometry of the scene. So, we will start with rendering with triangles using ray casting in this section.

Recent AMD Radeon HIP-compatible GPUs since RDNA2 has the capability to accelerate the ray intersections; however, developers have to use device-specific low-level APIs and design the algorithm carefully to achieve maximum ray tracing performance. Thus, we generally recommend using the HIPRT SDK, which we introduce later in this technical report. Although using such a vendor-provided SDK is often the optimal choice for most user applications in terms of performance, numerical robustness, and ease of implementation, use of the SDK as a black box may end up with inefficient usage or make debugging harder. Thus, we first explain ray tracing without using these APIs so readers can understand the algorithms behind these APIs, so that we believe this helps readers to use the APIs properly, understand the limitations.

3.1 Ray-Triangle Intersection

The intersection of a ray and a triangle can be split into two steps. The first step is to find the intersection with a plane that the triangle belongs to, and the second step is to check whether the point is inside the triangle. Let us describe a ray as a parametric equation:

$$\mathbf{R}(t) = \mathbf{o} + t\mathbf{d}, \tag{1}$$

where t is a parameter (i.e., distance) of the ray, $\mathbf{o}$ is a ray origin, $\mathbf{d}$ is a ray direction. Given triangle vertices $(\mathbf{v_0}, \mathbf{v_1}, \mathbf{v_2})$, the normal of the triangle $\mathbf{n}$, we can describe the problem by the following

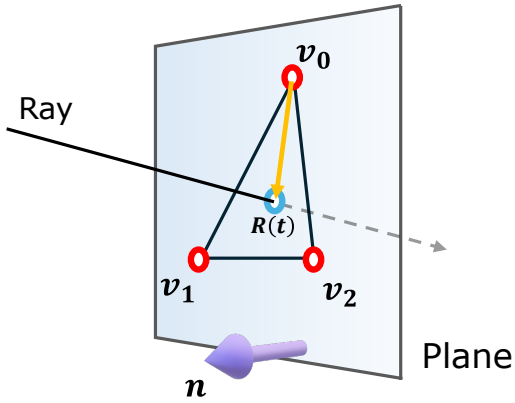


Fig. 3. To find the intersection with the plane of the triangle, we are looking for a value of t such t that $\mathbf{R}(t) - \mathbf{v_0}$ is perpendicular to the normal of the triangle.

equation:

$$\mathbf{n} \cdot (\mathbf{R}(t) - \mathbf{v_0}) = 0. \tag{2}$$

The intersection points is inside the triangle plane, and thus the vector $\mathbf{R}(t) - \mathbf{v_0}$ belongs to the plane as well. The dot product corresponds to a cosine of the angle between two vectors. If the the dot product is zero, two vectors are perpendicular. To satisfy the equality, point $\mathbf{R}(t)$ must lie on the plane, and thus it is the intersection point. Figure 3 illustrates the geometric interpretation of this equation. By plugging Equation 1 into Equation 2, we can explicitly express parameter t that satisfy the condition:

$$t = \frac{\mathbf{n} \cdot (\mathbf{v_0} - \mathbf{o})}{\mathbf{n} \cdot \mathbf{d}}. \tag{3}$$

Since we already know the hit point $\mathbf{R}(t)$ in the plane, we can check if the point is inside the triangle. This test can be described as three edge tests as follows:

$$\begin{aligned} 0 &\leq \mathbf{n} \cdot ((\mathbf{v_1} - \mathbf{v_0}) \times (\mathbf{R}(t) - \mathbf{v_0})), \\ 0 &\leq \mathbf{n} \cdot ((\mathbf{v_2} - \mathbf{v_1}) \times (\mathbf{R}(t) - \mathbf{v_1})), \\ 0 &\leq \mathbf{n} \cdot ((\mathbf{v_0} - \mathbf{v_2}) \times (\mathbf{R}(t) - \mathbf{v_2})), \end{aligned} \tag{4}$$

where $\times$ denotes the cross product of three-dimensional vectors and $\mathbf{n} = (\mathbf{v_1} - \mathbf{v_0}) \times (\mathbf{v_2} - \mathbf{v_1})$. Figure 4 shows one of the edge tests. Listing 4 shows an implementation of the intersection test.

Listing 4. Ray-triangle intersection algorithm.

```
__device__ bool intersectRayTriangle(float& tOut, const float3& rayOrigin, const
    float3& rayDirection, const float3& v0, const float3& v1, const float3& v2)
{
    const float3 e0 = v1 - v0;
    const float3 e1 = v2 - v1;
    const float3 e2 = v0 - v2;
    const float3 n = cross(e0, e1);
    const float t = dot(v0 - rayOrigin, n) / dot(n, rayDirection);
    if (MIN_T <= t && t <= MAX_T)
    {
        const float3 p = rayOrigin + rayDirection * t;
```

```
        const float a = dot(n, cross(e0, p - v0));
        const float b = dot(n, cross(e1, p - v1));
        const float c = dot(n, cross(e2, p - v2));
        if (a < 0.0f || b < 0.0f || c < 0.0f)
            return false;
        tOut = t;
        return true;
    }
    return false;
}
```

3.2 Camera Model

To render an image, we have to simulate how a real-world camera works. The simplest camera system is a pinhole camera: a box with a small hole on one of the sides and a film on the opposite side in the box. The small hole can work as a lens, such that the light coming from the outside is projected onto the film, forming an image. Figure 5 illustrates how a pinhole camera works.

In computer graphics, we often use an even simpler model of the real pinhole camera. A virtual film is placed in front of the lens, and the film is split into pixels (see Figure 6a). In this model, we shoot a ray from the lens through a pixel into the virtual scene, and find an intersection with the ray in order to calculate the color of the pixel. We introduce a parameterization in Figure 6b, where the lens location is as the origin, the size of the film as the up and right vectors, and the forward direction is implicitly represented as the cross product of the up and right vectors. Note that the distance from the origin and the virtual film is 1. Listing 5 shows how to generate a camera ray given camera parameters.

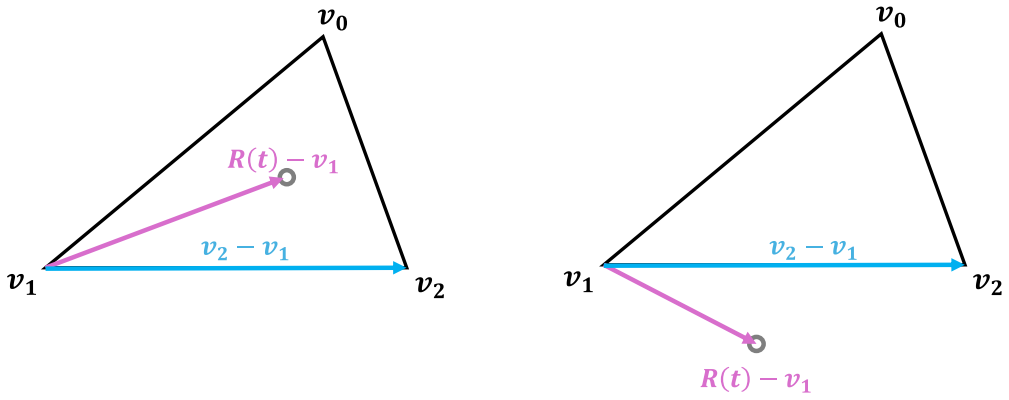


Fig. 4. The left image shows that the hit point is inside with respect to the edge $\mathbf{v}_2, \mathbf{v}_1$, which can be conditioned by $0 \leq \mathbf{n} \cdot ((\mathbf{v}_2 - \mathbf{v}_1) \times (\mathbf{R}(t) - \mathbf{v}_1))$ while the right shows the hit point is outside against the edge.

Listing 5. Generating a camera (primary) ray passing throught a pixel on the virtual screen; u and v are normalized coordinate on the screen.

```
struct RayGenerator
{
    float3 m_origin;
    float3 m_right;
    float3 m_up;
    __device__ void getPrimaryRay(float3& ro, float3& rd, float u, float v)
    {
        float3 from = m_origin;
        float3 forward = normalize(cross(m_up, m_right));
        float3 to = m_origin + forward + lerp(-m_right, m_right, u) +
                    lerp(m_up, -m_up, v);
        ro = from;
        rd = normalize(to - from);
    }
};
```

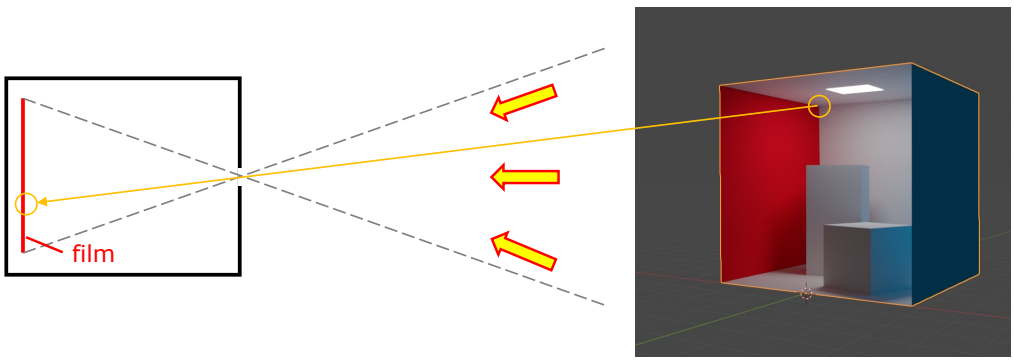
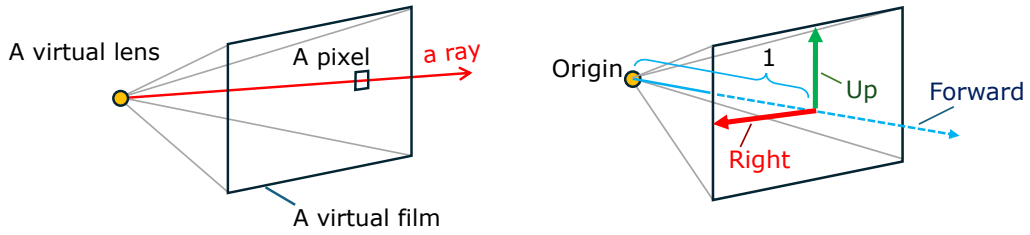


Fig. 5. Illustration of pinhole camera: light rays pass through the pinhole, projecting on the film on the opposite side.



(a) A simplified pinhole camera model

(b) A parameterization of the camera

Fig. 6. Pinhole camera model commonly used in computer graphics.

3.3 Rendering Triangles

Each such ray passing through a pixel can be processed independently, and thus multiple rays can be processed in parallel, which makes it suitable to execute on the GPU. Listing 6 shows a HIP implementation of rendering a single triangle.

Listing 6. Rendering a single triangle using ray tracing.

```
__global__ void RenderTriangleKernel(uint8_t* pixels, RayGenerator rayGen, const
    int width, const int height, const float3 v0, const float3 v1, const float3 v2
    )
{
  const int tid = threadIdx.x + blockDim.x * blockIdx.x;
  if (width * height <= tid) { return; }
  const int xi = tid % width;
  const int yi = tid / width;
  float3 rayOrigin, rayDirection;
  rayGen.getPrimaryRay(rayOrigin, rayDirection,
    static_cast<float>(xi) / width, static_cast<float>(yi) / height);
  const int pixelIdx = xi + (height - yi - 1) * width;
  float t;
  bool hit = intersectRayTriangle(t, rayOrigin, rayOrigin, v0, v1, v2);
  pixels[pixelIdx * 4 + 0] = (hit)?R_triangle:32;
  pixels[pixelIdx * 4 + 1] = (hit)?G_triangle:32;
  pixels[pixelIdx * 4 + 2] = (hit)?B_triangle:32;
  pixels[pixelIdx * 4 + 3] = 255;
}
```

The code in Listing 6 renders of a single triangle; nonetheless, scenes typically consist of many triangles. Since light hit the closest surface along the ray, we do not see objects behind of other (opaque) objects, and thus, we need to find the closest hit point among the triangles to simulate the light behavior. Listing 7 shows an implementation of a naïve linear search, rendering multiple triangles. Figure 7a shows an image rendered by ray tracing using the closes hit.

Listing 7. Rendering multiple triangles using ray tracing.

```
  // Generate camera rays, etc.
  ...
  float minT = FLT_MAX;
  int triIdx = -1;
  for (int i = 0; i < triangles.size(); i++)
  {
```

```
    Triangle tri = triangles[i];
    float t;
    if (intersectRayTriangle(t, rayOrigin,
      rayDirection, tri.vtx[0], tri.vtx[1], tri.vtx[2]))
    {
      if (t < minT)
      {
        triIdx = i; minT = t;
      }
    }
  }
  // Store the results.
  ...
}
```

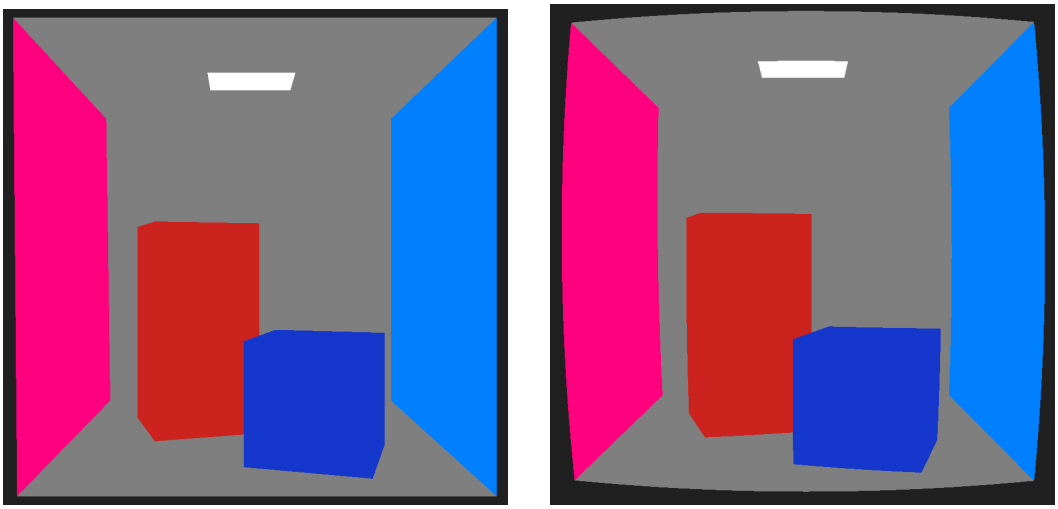
3.4 Bonus: Lens Distortion

A pinhole camera cannot model a real-world camera perfectly due to the shape and curvature of the lens, causing *radial distortion* (e.g., straight lines are not perfectly straight). We can model a simple radial distortion as a mapping from a distorted location on the virtual film to a non-distorted space, where we can trace the corresponding ray as with a standard pinhole camera. We model the mapping as $\mathbf{p}_{non-distorted} = \frac{\mathbf{p}_{distorted}}{1+c}$, where c is a coefficient proportional to the squared distance from the center of the screen $c = d\|\mathbf{p}_{distorted}\|^2$, where d is a distortion parameter. The implementation is shown in Listing 8 and the rendered result is depicted in Figure 7b.

Listing 8. Primary rays taking into account radial distortion of the lens.

```
struct RayGenerator
{
    float3 m_origin;
    float3 m_right;
    float3 m_up;
    __device__ void getPrimaryRay(float3& ro, float3& rd, float u, float v)
    {
        float3 from = m_origin;
        float3 forward = normalize(cross(m_up, m_right));

        float3 X = lerp(-m_right, m_right, u);
        float3 Y = lerp(m_up, -m_up, v);
        float3 p_distorted = X + Y;
        float r2 = dot(p_distorted, p_distorted);
        float d = -0.5f;
        float c = d * r2;
        float3 to = m_origin + forward + (X + Y) / (1.0f + c);
        ro = from;
        rd = normalize(to - from);
    }
};
```



(a) A pinhole camera (b) A pinhole camera with radial distortion

Fig. 7. Cornell box rendered by ray tracing using the pinhole camera model and the camera model with radial distortion.

4 AMBIENT OCCLUSION

In the previous section, only the visibility from the camera is taken into account in the rendering. Since simulating full light transport is complex, let us start with a simple effect known as ambient occlusion, approximating indirect lighting by relative occlusion by geometry in a proximity of the shading point. Note that indirect lighting is lighting that does not come directly from a light source, reaching the shading points indirectly through surface interactions (e.g., refraction or transmission). The occlusion can be measured by shooting rays from the shading point. Figure 8 shows an example of ambient occlusion calculation with a limited number of rays.

4.1 Cosine Law and Cosine-Weighted Sampling

In reality, incoming light does not illuminate the shading point equally from all directions. The amount of light reaching the shading point is proportional to the cosine of the angle between the direction and the normal. This is known as *Lambert's cosine law*, which is illustrated in Figure 9. To take this into account, we need not only to multiply the relative occlusion by the cosine, but also to normalize the result by π such that a completely unoccluded shading point corresponds to 1.

A common practice is to sample directions randomly, unlike uniform intervals in Figure 8. This is because such a regular sampling pattern produces visually disturbing correlated patterns in the rendered images. Due to the cosine term, it is beneficial to sample the direction according to a distribution proportional to the cosine of the angle instead of a uniformly distributed direction (we will discuss this in more detail in Section 5). One of the simplest ways to realize this is to project a uniform distribution in a circle to a hemisphere. This works because of the area relationship between the circle and the hemisphere as shown in Figure 10, where the relationship between a small area on the hemisphere ΔA and its projection $\Delta A_\perp$ on the bottom circle can be described as $\Delta A_\perp = \Delta A \cos\theta$. As the area on the circle is stretched by $\frac{1}{\cos\theta}$ projecting to the hemisphere, the density of the distribution on the hemisphere is scaled by $\cos\theta$. Changing the distribution using geometric relationships is often used to achieve the desired distribution (i.e.,

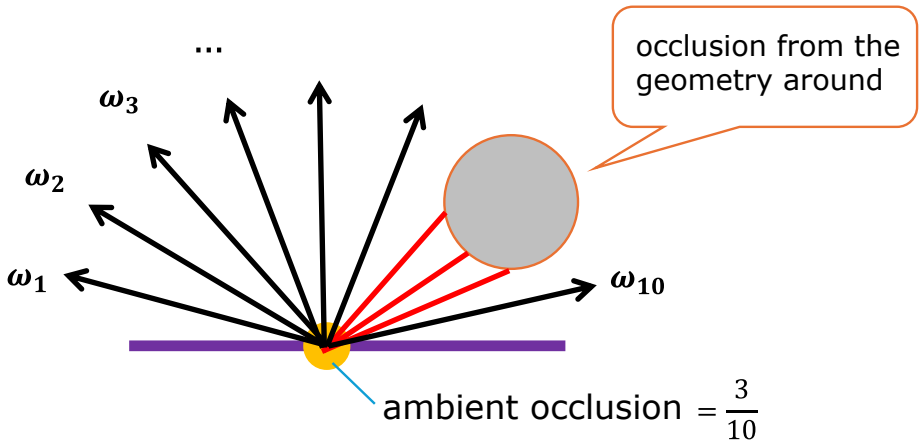


Fig. 8. An illustration how the incoming light is occluded by the geometry in the proximity of the shading point.

place samples proportional to the cosine value). For uniform distribution on a circle, we can use $\phi = 2\pi\xi_0, r = \sqrt{\xi_1}$ in the polar coordinate system, where ξ_0, ξ_1 are uniform random numbers in $[0, 1]$, and $x = r\cos\phi, z = r\sin\phi$. The square root here works as a cancellation of the radial compression introduced by the variable transformation from a unit square to a unit circle. Listing 9 shows an implementation of the cosine-weighted sampling. Since the hemisphere used in the implementation is centered around Y axis, we need to transform the the coordinates system of the shading point. We need to compute a basis of the coordinate system for the the transformation consisting of three vectors: normal, tangent, and bi-tangent. We have a normal vector, the tangent cane be pre-calculated by modeling software, but we can also use any of two vertices of the triangle, and the bi-tangent can be computed as a cross product of the normal and tangent vectors.

4.2 Practical Consideration

By using the cosine-weighted hemisphere sampling described above, Listing 10 shows the example code for the ambient occlusion calculation, and the result is shown in Figure 11a. Note that we add a small offset to the ray origin based on the normal at the point to avoid self-intersection with the base triangle. Ambient occlusion does not strictly follow physical light behavior and ignores multiple-bounce lighting effects, but it is far more interesting than the image we rendered in Figure 7a and it resembles real-world occlusion. As it can be precomputed and baked into a texture, it has been used as a cheap approximation of shadows in real-time applications. In addition, ambient occlusion can be used for arbitrary non-photorealistic rendering. For example, Figure 11b shows a simple false color rendering by mapping the ambient occlusion value into a color table.

We used ray tracing to find the closest hit along the ray, considering any intersection along the ray. In practise, we limit the length of a ray from the shading point to control the locality of the ambient occlusion. Thus, we do not need to test triangles behind the second point, and we can also stop immediately after finding first intersection (not necessarily the closest one) as we are interested only in occlusion. Ray tracing frameworks such as HIPRT support the maximum ray

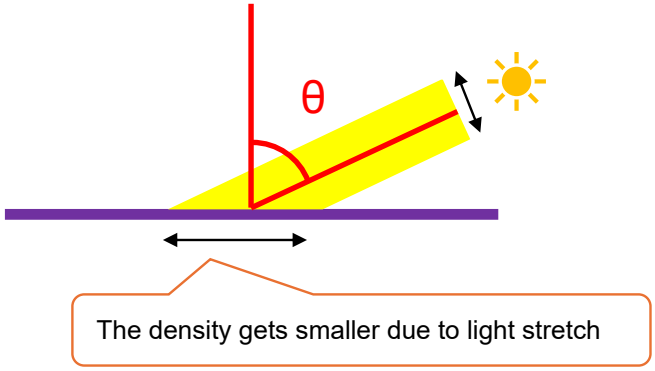


Fig. 9. An illustration of Lambert's cosine law: the incoming light is attenuated by the cosine term as the density of photons is smaller with larger angles.

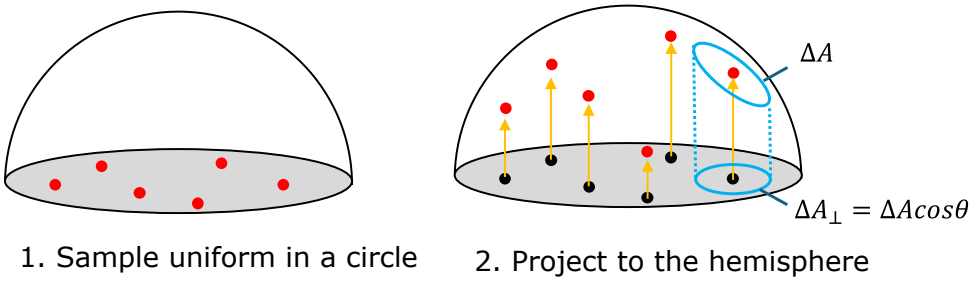


Fig. 10. An illustration of cosine-weighted sampling: we first uniformly sample points in the circle, and then project them to the hemisphere to get the random directions.

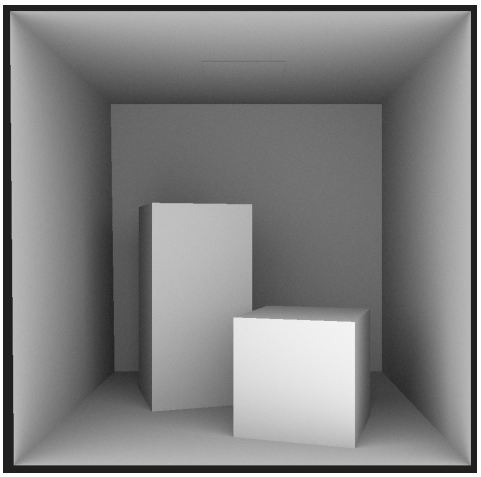
length and provide optimized any-hit kernels that can be used for such tests. Figure 12 shows the effect of the ray length in ambient occlusion.

Listing 9. Cosine-weighted hemisphere sampling.

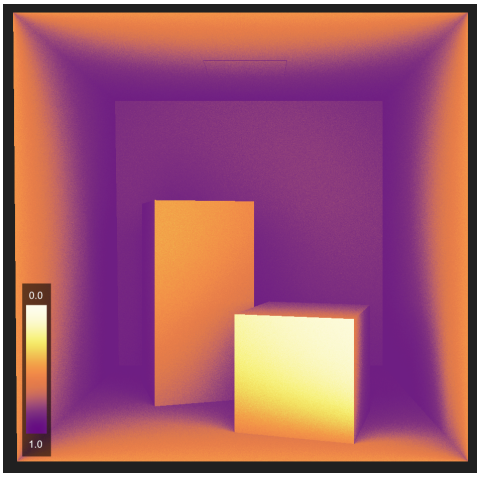
```
float3 sampleHemisphere(float xi_0, float xi_1)
{
  float phi = xi_0 * 2.0f * PI;
  float r = sqrtf(xi_1);
  // uniform in a circle
  float x = cosf(phi) * r;
  float z = sinf(phi) * r;
  // project to a hemisphere
  float y = sqrtf(fmax(1.0f - r * r, 0.0f));
  return {x, y, z};
}
```

Listing 10. Calculation of ambient occlusion.

```
const float3 p = rayOrigin + rayDirection * t;
const float3 aoRayOrigin = p + n * 0.0001f;
constexpr int N_RAYS = 1024;
int nOcclusions = 0;
for (int i = 0; i < N_RAYS; i++)
{
  float3 s = sampleHemisphere(random.uniformf(), random.uniformf());
  // The hemisphere is Y-up. Transform the Y axis to the normal
  float3 aoRayDirection = tangent0 * s.x + tangent1 * s.z + n * s.y;
  Intersection aoIsect;
  if (closesetHit(aoIsect, aoRayOrigin, aoRayDirection, triangles))
  {
    nOcclusions++;
  }
}
const float ao = static_cast<float>(nOcclusions) / N_RAYS;
```



(a) Ambient occlusion in grayscale



(b) A false-color rendering of ambient occlusion with a gradient look-up table.

Fig. 11. Visualizations of Ambient Occlusion.

(a) 0.1 (b) 0.3 (c) 2.0

Fig. 12. Ambient Occlusion with different ray lengths.

5 PATH TRACING

Path tracing [3] is one of the most popular techniques to render photorealistic images. It is used in a wide range of applications, from offline movie production to real-time rendering in games. Producing a photorealistic image means simulating how light behaves in the real world. In path tracing, we simulate the optical paths that come to the camera from light sources. This helps us to compute how much light arrives at each pixel in the final image. In the real world, there are infinitely many possible paths that light can take before it reaches the camera. Simulating all of them is computationally impossible.

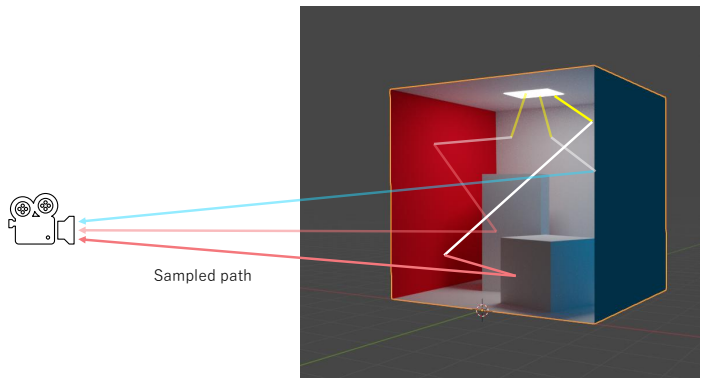


Fig. 13. A visualization of many light paths from a light source to the camera.

To address this, path tracing uses random sampling to generate a limited number of light paths (Figure 13). This technique is called *Monte Carlo ray tracing*. By averaging the results of many random paths, we can estimate the amount of light reaching each pixel. Ambient occlusion in the previous section is one example of Monte Carlo ray tracing. Because the method relies on random numbers, the resulting image contains noise. As we increase the number of samples (i.e., light paths), the noise decreases, and the image becomes clearer (see Figure 14). In practical applications like movies and games, reducing noise while keeping the computation time short is a major challenge. Although we do not cover them in this section, many advanced techniques have been developed to tackle this challenge. [3–6]

(a) 16 samples (b) 128 samples (c) 1024 samples

Fig. 14. A comparison of images with different sample counts.

5.1 Algorithm

In the real world, light travels from light sources to the camera. However, in path-tracing, we construct light paths in the opposite direction - from the camera toward the light sources. The reason for this is efficiency. If we trace paths from the light source, most of them would never reach the camera. Starting from the camera, we only trace paths that contribute to the image.

We use ray-tracing to construct these paths. For each pixel, we first generate a ray that starts from the camera and passes through the pixel. We trace this ray into the scene and find its intersection with objects in the same manner as we did in the case of ambient occlusion. At the intersection point, we randomly sample a new direction and generate a new ray from that point. This process simulates reflection. We repeat this process, tracing the new ray, sampling a new direction, and so on, until the ray eventually hits a light source. When a ray reaches a light source, we add the light contribution to the pixel's color. Figure 15 illustrates this algorithm.

In the real world, when light hits a surface, some of the energy is absorbed, and the rest is reflected. To simulate this behavior, the path-tracing algorithm multiplies the reflectance of the surface by the light emission. The accumulated product of the reflectance along the ray is called *throughput*. When a ray reaches a light source, we multiply the ray's throughput by the light emission and accumulate it in the pixel.

Since there can be multiple bounces before hitting a light source, we need to take all of them into account. We can denote the contribution for a single bounce case as $R_0 \times E_0$, where R_i is the reflectance, E_i is the light emission, and the subscripts are the number of bounces. Also, we can write two-bounce case as $R_0 \times R_1 \times E_1$. Thus, we can sum them up to get the total contribution $L(i)$ for the number of bounces i as follows:

$$L(i) = E_0 + (R_0 \times E_1) + (R_0 \times R_1 \times E_2) + \ldots + (R_0 \times R_1 \times \ldots \times R_{i-1} \times E_i).$$

We can also simplify products of R by throughput T_i, the accumulated product of the reflectance $T_i = R_0 \times R_1 \times \ldots \times R_{i-1}$. Thus, we can simplify it as follows:

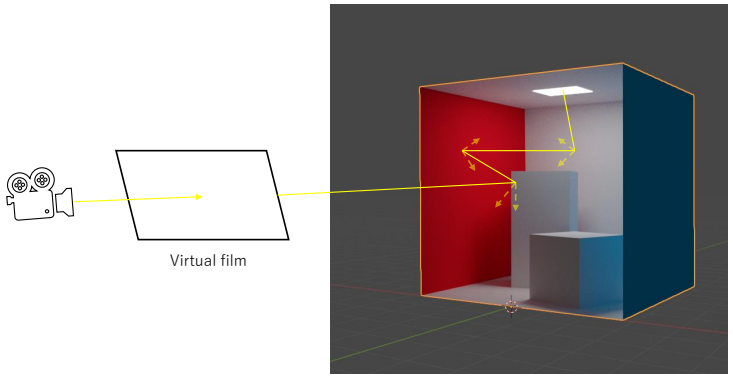


Fig. 15. Illustration of the path tracing algorithm.

$$L(i) = T_0 \times E_0 + T_1 \times E_1 + T_2 \times E_2 + \ldots T_i \times E_i = \sum_{k=0}^{i} T_k E_k.$$

This lead us to a simple for loop to solve the light contribution as shown below.

Listing 11. A high-level Path tracing algorithm.

```
float3 radiance = {0.0f, 0.0f, 0.0f};
float3 throughput = {1.0f, 1.0f, 1.0f};
for (int depth = 0; depth < options.maxDepth ; ++depth)
{
    radiance += throughput * emissive(depth);
    throughput *= reflectance(depth);
}
```

Listing 12 shows a complete example code of this algorithm, including ray tracing. Note that the idea and the structure of the algorithm are the same as the code above.

Listing 12. Path tracing algorithm.

```
Ray ray = makeRay(ro, rd);
float3 radiance = {0.0f, 0.0f, 0.0f};
float3 throughput = {1.0f, 1.0f, 1.0f};
for (int depth = 0; depth < options.maxDepth; ++depth)
{
  Intersection isect;
  if (!raytrace(ray, hiprtGeom, isect))
  {
    // hit nothing
    radiance += throughput * options.skyColor;
    break;
  }
  Triangle hitTriangle = triangles[isect.index];
  if (hasEmission(hitTriangle))
  {
    // hit light source
    radiance += throughput * triangles[isect.index].emissive;
    break;
  }
```

```
  // Evaluate hit position and hit normal
  SurfaceInfo surf = makeSurfaceInfo(ray, isect, triangles);
  float3 wo;
  {
    TangentBasis basis =
      makeTangentBasis(surf.n, isect.index, triangles);
    // Sample next ray direction in tangent space
    float3 woLocal = sampleHemisphere(
      random.uniformf(), random.uniformf());
    // Transform direction from tangent space to world space
    wo = localToWorld(wo_local, basis);
  }
  // Update throughput
  throughput *= hitTriangle.color;
  // Generate next ray
  ray = makeRay(offsetRayPosition(surf.p, surf.n), wo);
}

// Write results to the accumulation buffer
accumulation[pixelIdx] += {radiance.x, radiance.y, radiance.z, 1.0f};
```

Figure 14 shows rendering results using path tracing with different sample counts. Fewer samples result in noisier images. The amount of noise depends on how likely the randomly generated light paths are to reach a light source. If most of the sampled paths miss the light source, the estimation becomes highly variable, leading to visible noise. To reduce noise, it is important to sample light paths that are more likely to reach the light source. This idea is known as *importance sampling*. In the next section, we introduce one such importance sampling technique called *Next event estimation*.

5.2 Next Event Estimation

One of the main issues with basic path tracing is that rays are sampled randomly without considering the positions of light sources. As a result, many sampled paths fail to reach any light source, leading to an inefficient computation which results in high variance in the final image (Figure 16). Next event estimation [3; 4] addresses this problem by explicitly taking the positions of light sources into account during sampling.

At each surface intersection, next event estimation randomly selects one of the light sources in the scene. Once a light source is selected, a point on the light source is sampled. Then, a ray is generated to connect the intersection point and the sampled point on the light source. This ray

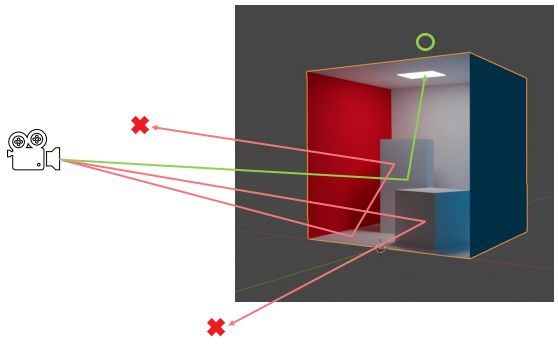


Fig. 16. Failure cases of basic path tracing.

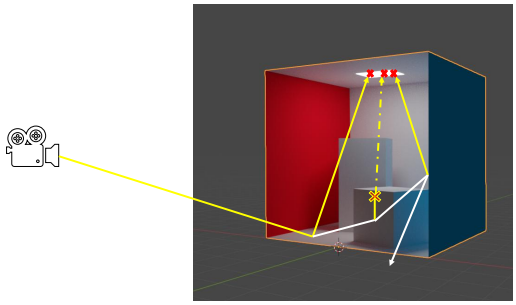


Fig. 17. Next event estimation

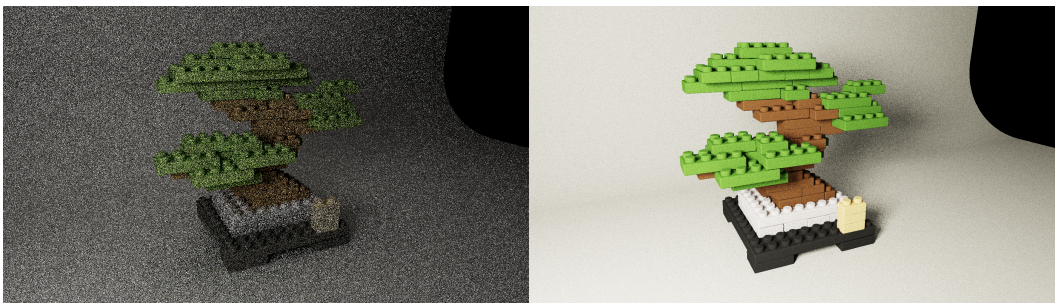
is called a *shadow ray*. If a shadow ray is not blocked by any objects, we accumulate the light contribution to the pixel. Whether a shadow ray is blocked or not is determined by ray tracing (similarly to ambient occlusion). This is the basic idea behind the next event estimation algorithm. Figure 17 illustrates this idea. Listing 13 is an example code of the next event estimation algorithm.

Listing 13. Next event estimation algorithm.

```
// ...
for (int depth = 0; depth < options.maxDepth; ++depth)
{
  // raytrace
  // ...
  Triangle hitTriangle = triangles[isect.index];
  if (hasEmission(hitTriangle))
  {
    // hit light source
    if (depth == 0) {radiance += throughput * hitTriangle.emissive;}
    break;
  }
  // Evaluate hit position and hit normal
  // ...
  // Sample a position on the light source
  LightSample lightSample =
    sampleLight(triangles, lights, random.uniformf(),
                random.uniformf(), random.uniformf());
  // Create a path between the hit position and the light source
  {
     Triangle lightTriangle = triangles[light_sample.index];
     const float V =
       checkVisibility(surf.p, surf.n, lightSample.p, hiprtGeom);
     const float3 brdf = 1.0f / PI * hitTriangle.color;
     const float G =
       geometryTerm(surf.p, surf.n, lightSample.p, lightSample.n);
     const float lightPdf =
       1.0f / lights.size() * 1.0f / calculateArea(lightTriangle);
     radiance +=
       throughput * brdf * G * V * lightTriangle.emissive / lightPdf;
  }
  // Sample next ray direction
  // ...
  // Update throughput
  // ...
```

```
36   // Generate next ray
37   // ...
38 }
39
40 // Write results to the accumulation buffer
41 // ...
```

The next event estimation algorithm is quite similar to basic path tracing, but there are a few important differences. First, in Next event estimation, the contribution from rays that hit a light source directly is not added — except when the ray is at depth 0. This is to avoid counting the same light path twice. Second, the way we compute the contribution from the connection between the surface point and the sampled point on the light is different. As shown in the code example, several terms are multiplied with the light intensity, including the *bidirectional reflectance distribution function (BRDF)*, *geometry term*, and *probability density function (PDF)* of the sampled ray. These terms are essential to make the estimation statistically unbiased. If we omit them, the rendered result will differ from basic path tracing. Although detailed explanations of these terms are beyond the scope of this book, we encourage interested readers to read [7] for more information. Figure 18 shows rendering results of Next event estimation. Compared to basic path tracing with the same number of samples, the noise is significantly reduced. As this example demonstrates, designing better methods to sample light paths is crucial for reducing noise in the rendered image.



(a) Path tracing (b) Next event estimation

Fig. 18. A comparison of Path tracing and Next event estimation in 16 samples.

6 BOUNDING VOLUME HIERARCHY (BVH)

In the previous sections, we tested all triangles sequentially (i.e., linear time complexity) to find the intersections with a given ray. In practice, we deal with scenes of significantly higher complexity, consisting of millions of triangles (or other geometric primitives). Testing all triangles in a linear fashion becomes practically infeasible even on highly parallel GPUs already for scenes of moderate complexity. Therefore, significant effort has been devoted to reducing the number of intersection tests. The core idea is to arrange the triangles into spatial data structures that exploit the spatial proximity of the triangles such that we can efficiently cull parts of the scene (e.g., a subset of triangles) that are certainly not intersected. In this section, we focus on the bounding volume hierarchy (BVH), one of the most widely adopted acceleration data structures in modern ray tracing frameworks.

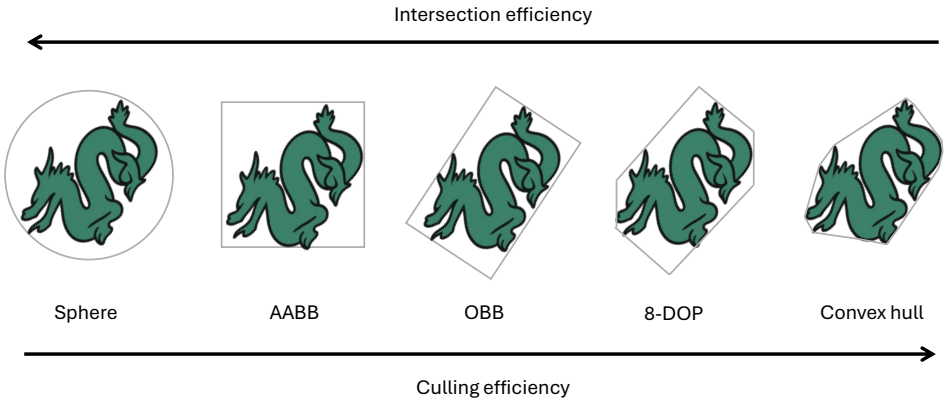


Fig. 19. An example of commonly used bounding volumes for ray tracing, balancing trade-off between culling efficiency (tightness) and the intersection efficiency: sphere, axis-aligned bounding box (AABB), oriented bounding box (OBB), discrete oriented polytope (DOP), and convex hull.

6.1 Bounding Volumes

The idea is to enclose scene objects (or any subset of triangles) within simpler bounding volumes that can be easily tested for an intersection. If a ray does not intersect the bounding volume, we know that there is no intersection with the objects inside, and thus we do not need to test the objects inside. There are various types of bounding volumes, typically balancing culling efficiency (how tight the bounding volume is) and intersection efficiency (how quickly we can compute the intersection). Tighter bounding volumes can better cull rays thanks to reduced empty space, but the intersection test is typically more complex. For ray tracing, we use axis-aligned bounding boxes (AABBs), defined by minimum and maximum points. For elongated diagonal triangles, it may be beneficial to use oriented bounding boxes (represented as AABBs with additional rotation matrices) (OBBs), which better fit these non-axis aligned shape.

6.2 Bounding Volume Hierarchy (BVH)

A drawback of bounding volumes is that if we hit a bounding volume, we still need to compute intersections with all objects inside. Bounding volumes can be nested in a hierarchical manner to form a rooted tree with bounding volumes in the internal nodes and triangles in the leaf nodes [8]. We can then find the intersection by traversing the hierarchy from the root, skipping the nodes that are not intersected. While the logarithmic complexity is not guaranteed in general, depending on how the tree is balanced, the complexity is significantly reduced in practice. Contemporary ray tracing frameworks use wide trees with a maximum branching factor of 4 or 8.

BVH Construction. To be able to find the intersection, we need to construct the BVH first. There are two main construction approaches: top-down (splitting) [9] and bottom-up (clustering) [10]. The former approach starts with all triangles, and recursively splits them into disjoint subsets until a termination criterion (e.g., the number of triangles in a subset is less than a predefined threshold) is satisfied. The latter approach considers each triangle as a cluster, then iteratively merges the clusters until only one clusters remains, which corresponds to the root node. Notice that there are many possible BVHs for a given scene or model. For example, there are many ways to split the triangles

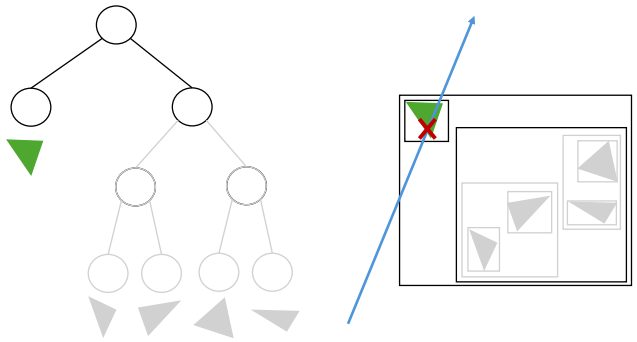


Fig. 20. An example a simple bounding volume hierarchy (BVH) using axis-aligned bounding boxes (AABBs). To find the closest intersection (the red cross), we need to test three AABBs and one triangle (the gray part is culled off).

during the top-down construction or to merge the clusters during the bottom-up construction. These local decisions have an impact on culling efficiency, which is inversely proportional to a cost function defined as a sum of surface areas of the bounding volumes in the internal nodes. BVH construction algorithms aim to minimize the surface areas and the construction times, preferring one of these criteria depending on a particular application [8]. The construction speed is critical for dynamic geometry, where once the underlying triangles change, the BVH becomes invalid, and thus must be reconstructed. An alternative approach is to keep the topology of the BVH the same and fit the bounding volumes to the current geometry, which can be done efficiently in a bottom-up manner. Depending on the bounding volume type, we can compute new bounding volumes simply from the child bounding volumes (e.g., for AABBs). A caveat is that the culling efficiency may degrade if the geometry is very different than the original geometry for which the BVH was initially constructed.

BVH Traversal. Once the BVH is constructed, it is relatively straightforward to use it to find the intersections. To traverse the BVH, we typically employ a stack to store the nodes to be tested, starting by pushing the root node onto the stack. In each step, we pop a node from the top of the stack and test it, if the bounding volume of the node is intersected, we either push its children onto the stack (in the case of an internal node), or test triangles inside (in the case of a leaf node). We repeat this until the stack is empty. Using this procedure, we can find both the nearest intersection, any intersection, or all intersections.

Two-Level Hierarchy and Instancing. In practice, it is beneficial to arrange the scene into two levels [11], where the bottom level consists of BVHs of individual scene objects (bottom-level acceleration structure - BLAS) and the top-level BVH (top-level acceleration structure - TLAS) is built over these BVHs such that the leaf nodes of the top BVH contain references to the bottom level BVHs and an affine transformation. There are two advantages of this approach. This approach allows instancing of the bottom level objects. We reference the same bottom-level BVH multiple times in the top-level BVH with different transformations (e.g., a classroom with multiple instances of desks and chairs), which reduces the memory usage. Moreover, this approach enables rigid animations

by changing the transformations such that only the top-level BVH has to be reconstructed, which significantly reduces the update time in real-time applications. A drawback is that the ray traversal becomes more complicated. When we encounter an instance node during the traversal, we need to transform the ray into the local object space using the inverse transformation.

7 HIPRT: HIP RAY TRACING API

While implementing a simple variant of BVH is relatively straightforward, optimizing the code for high performance is very challenging. HIPRT [12] is an open-source ray tracing framework written in HIP, implementing the BVH construction, traversal, and other essential features required in modern renderers. HIPRT is optimized for AMD GPUs (including MI series), utilizing specialized ray tracing hardware units on RDNA 2 and later architectures. In this technical report, we briefly introduce the HIPRT API, illustrating how to build a two-level hierarchy on a simple example.

7.1 Context Initialization

Note that HIPRT performs all computation entirely on the GPU, and thus all input buffers in the following code are supposed to be allocated on the device. We start with the HIPRT context initialization and setting the logging level that may be helpful for debugging (see Listing 14).

Listing 14. Context initialization and log level setting.

```
#include <hiprt/hiprt.h>

hiprtContextCreationInput hiprtCtxInput;
hiprtCtxInput.deviceType = hiprtDeviceAMD;
hiprtCtxInput.ctxt = oroGetRawCtx(ctx);
hiprtCtxInput.device = oroGetRawDevice(device);

hiprtContext hiprtCtx;
hiprtCreateContext(HIPRT_API_VERSION, hiprtCtxInput, hiprtCtx);

hiprtSetLogLevel(hiprtLogLevelError | hiprtLogLevelWarn);
```

7.2 Geometry Construction

We build a bottom-level BVH for a triangle mesh. We first initialize the HIPRT triangle mesh, geometry build input, and build options structures (see Listing 15). In the following example, we assume indexed geometry with triangles defined in a consecutive array of `uint3` structures and vertices are stored in a consecutive array of `float3` structures. In the build options, we can specify the construction algorithm and a few other options. In this case, we use the balanced option that provides very good culling efficiency and the build is still very fast (the other two options are fast and high-quality).

Listing 15. Triangle mesh and geometry build input.

```
hiprtTriangleMeshPrimitive mesh{};
mesh.triangleCount = /* the number of triangles */;
mesh.triangleStride = sizeof(uint3);
mesh.triangleIndices = /* a device pointer to an array of uint3 */;
mesh.vertexCount  = /* the number of vertices */;
mesh.vertexStride = sizeof(float3);
mesh.vertices = /* a device pointer to an array of float3 */;

hiprtGeometryBuildInput geomInput{};
geomInput.type = hiprtPrimitiveTypeTriangleMesh;
```

```
geomInput.primitive.triangleMesh = mesh;

hiprtBuildOptions options{};
options.buildFlags = hiprtBuildFlagBitPreferBalancedBuild;
```

The construction requires a temporary space for intermediate computations. Based on the mesh size and other options, we query the size of this space and allocate the corresponding temporary buffer on the device (see Listing 16). This buffer can be reused for construction of other BVHs or any other computations on the user side.

Listing 16. Temporary buffer allocation.

```
size_t geomTempSize{};
hiprtGetGeometryBuildTemporaryBufferSize(hiprtCtx, geomInput, options,
    geomTempSize);

hiprtDevicePtr geomTempTris{};
oroMalloc(reinterpret_cast<oroDeviceptr*>(&geomTempTris), geomTempSize)
```

Last, we create and build the HIPRT geometry, which corresponds to the bottom-level BVH in the HIPRT terminology (see Listing 17).

Listing 17. Geometry creation and construction.

```
hiprtGeometry geomTris{};
hiprtCreateGeometry(hiprtCtx, geomInput, options, geomTris);
hiprtBuildGeometry(hiprtCtx, hiprtBuildOperationBuild,  geomInput, options,
    geomTempTris, 0 /* stream */, geomTris);
```

HIPRT supports custom geometric primitives such as spheres or curves. HIPRT is agnostic to a particular type of the primitive, taking a list of AABBs of these primitives provided by a user as an input (in contrast to the triangle mesh). In the example showed in Listing 18, we assume that each AABB is represented as two `float3` structures. The geometry type is a user defined value that will be later used to set up the custom intersection functions. The rest of the BVH construction remains the same as for triangles above.

Listing 18. AABB list and geometry build input.

```
hiprtAABBListPrimitive list{};
list.aabbCount = /* the number of custom primitives */;
list.aabbStride = 2 * sizeof(float3);
list.aabbs = /* a device pointer to an array of 2 * float3 */;;

hiprtGeometryBuildInput geomInput{};
geomInput.type = hiprtPrimitiveTypeAABBList;
geomInput.primitive.aabbList = list;
geomInput.geomType = 0;

hiprtDevicePtr geomTempCustoms{};
...
hiprtGeometry geomCustoms{};
hiprtBuildGeometry(hiprtCtx, hiprtBuildOperationBuild, geomInput, options,
    geomTempCustoms, 0 /* stream */, geomCustoms);
```

7.3 Scene Contruction

With constructed HIPRT geometries, we can build HIPRT scene, which corresponds to the top-level BVH. Similarly to geometries, we need to set up the scene build input, which takes an array of HIPRT instances. In Listing 19, we define two instance objects, and we use them for actual instancing, creating an array of the instance objects, where each object can be repeated arbitrary times.

Listing 19. Instance definition.

```
hiprtInstance instanceTris{};
instanceTris.type = hiprtInstanceTypeGeometry;
instanceTris.geometry = geomTris;

hiprtInstance instanceCustoms{};
instanceCustoms.type = hiprtInstanceTypeGeometry;
instanceCustoms.geometry = geomCustoms;

hiprtInstance instances[] = { instanceTris, instanceTris, ..., instanceCustoms,
    instanceCustoms };
constexpr size_t INSTANCE_COUNT = sizeof(instances) / sizeof(instances[0]);
```

Notice that we set the instance type. HIPRT supports multi-level instancing (i.e., instances of instances). In this technical report, we limit ourselves to two-level hierarchy, and thus we always use the geometry type. We also need to define transformation for each instance. In HIPRT, the instance transformation can be specified as a transform matrix (`hiprtFrameMatrix`) or by individual transformation components (`hiprtFrameSRT`). Listing 20 illustrates the latter case.

Listing 20. Instance tranformations - frames.

```
hiprtFrameSRT frames[INSTANCE_COUNT];
for (size_t i = 0; i < INSTANCE_COUNT; ++i)
{
  hiprtFrameSRT& frame = frames[i];
  frame.translation = /* translation component of the i-th frame */;
  frame.scale = /* scale component of the i-th frame */;
  frame.rotation = /* rotation component of the i-th frame */;
}
```

With the instances and transformations, we can finally set up the scene build input. In Listing 21, we allocate the device buffers for the instances and transformation, and copy the data to the device.

Listing 21. Scene build input and instance data transfer to a device.

```
hiprtSceneBuildInput sceneInput{};
sceneInput.instanceCount = INSTANCE_COUNT;
sceneInput.frameCount = INSTANCE_COUNT;

oroMalloc(reinterpret_cast<oroDeviceptr*>(&sceneInput.instances), INSTANCE_COUNT *
     sizeof(hiprtInstance));
oroMemcpyHtoD(reinterpret_cast<oroDeviceptr>(sceneInput.instances), instances,
    INSTANCE_COUNT * sizeof(hiprtInstance));

oroMalloc(reinterpret_cast<oroDeviceptr*>(&sceneInput.instanceFrames),
    INSTANCE_COUNT * sizeof(hiprtFrameSRT));
oroMemcpyHtoD(reinterpret_cast<oroDeviceptr>(sceneInput.instanceFrames), frames,
    INSTANCE_COUNT * sizeof(hiprtFrameSRT));
```

Once the scene build input is set up, we allocate the temporary buffer for the construction. Finally, we create and build the HIPRT scene (see Listing 22).

Listing 22. Scene creation and construction

```
size_t sceneTempSize{};
hiprtGetSceneBuildTemporaryBufferSize(hiprtCtx, sceneInput, options, sceneTempSize
    );

hiprtDevicePtr sceneTemp{};
oroMalloc(reinterpret_cast<oroDeviceptr*>(&sceneTemp), sceneTempSize)

hiprtScene scene{};
hiprtCreateScene(hiprtCtx, sceneInput, options, scene);
hiprtBuildScene(hiprtCtx, hiprtBuildOperationBuild, sceneInput, options, sceneTemp
    , 0 /* stream */, scene);
```

One could wonder why the frames are not stored in the same structure as instances. HIPRT supports instance-based motion blur such that one instance may have a range of frames. The ranges need to be specified as additional buffer in the scene build input (`instanceTransformHeaders`).

7.4 Custom Intersection

For the custom primitives, we need to set up the intersection function and also the data defining the custom primitives that are passed to the intersection function. To pass the data, we need to create the function table which is a 2D structure mapping a ray type and geometry type to the corresponding data in the intersection function. An example of this is showed in Listing 23. We assume a single ray type and geometry type, resulting in a single entry in the table.

Listing 23. Custom function table - initialization and data assignment.

```
hiprtFuncDataSet funcDataSet{};
funcDataSet.intersectFuncData = /* arbitrary device pointer */;

hiprtFuncTable funcTable{};
checkHiprt(hiprtCreateFuncTable(ctxt, 1 /* the number of geometry types */ , 1 /*
    the number of ray types */, funcTable));
checkHiprt(hiprtSetFuncTable(ctxt, funcTable, 0 /* geometry type */, 0 /* ray type
     */, funcDataSet));
```

The intersection function must be a device function with the signature showed in Listing 24, returning `true` in the case of intersection and `false` otherwise. The function takes a ray, the data we specified in the previous step, an optional payload, and the resulting hit structure.

Listing 24. Signature of a custom intersection function.

```
__device__ bool intersectCustom(const hiprtRay& ray, const void* data, void*
    payload, hiprtHit& hit) {...}
```

7.5 Traversal Stacks and Traversal Objects

Listing 25 shows how to use the constructed scene to find the intersections. We need to create a traversal object that can be used to find the intersection. HIPRT allows to customize the traversal stack. For high performance, we use the HIPRT global stack that combines local shared memory and global memory as a fallback. The global stack buffer must be big enough to accommodate stacks for all scheduled threads. Similarly, the shared memory buffer must be big enough for all threads in the block.

Listing 25. Global traversal stack setup shared and global buffers.

```
__shared__ uint32_t sharedStackCache[SHARED_STACK_SIZE * BLOCK_SIZE];
```

```
2
3  hiprtSharedStackBuffer sharedStackBuffer{};
4  sharedStackBuffer.stackSize = SHARED_STACK_SIZE;
5  sharedStackBuffer.stackData = sharedStackCache;
6
7  hiprtGlobalStackBuffer globalStackBuffer{};
8  globalStackBuffer.stackSize = /* global buffer size */;
9  globalStackBuffer.stackData = /* global buffer pointer */;
10
11 hiprtGlobalStack stack(globalStackBuffer, sharedStackBuffer);
12 hiprtEmptyInstanceStack instanceStack;
```

There are two main types of traversal objects: for closest intersection and for any/all intersections. Note that all intersections can be found by repeatedly calling `getNextHit()` on the traversal object. We can check the state of the traversal by calling `getCurrentState()` on the traversal object, which indicates whether all intersection have been found. The state of traversal is stored in the stack buffers. For example, if we reuse the buffers in the closest traversal object, we invalidate this state of the any-hit traversal object (as we did in the example above). An example of the traversal objects and using them to find the intersection is depicted in Listing 26.

Listing 26. Using the traversal object to find the intersections.

```
1  hiprtRay ray{};
2  ray.origin = /* ray origin */
3  ray.direction = /* ray direction */
4  ray.minT = /* min ray length */ MIN_T;
5  ray.maxT = /* max ray length */ MAX_T;
6
7  hiprtSceneTraversalAnyHitCustomStack<hiprtGlobalStack, hiprtEmptyInstanceStack>
       anyHitTr(scene, ray, stack, instanceStack, hiprtFullRayMask,
       hiprtTraversalHintDefault,
8      nullptr /* payload */, funcTable);
9  hiprtHit anyHit = anyHitTr.getNextHit();
10
11 hiprtSceneTraversalClosestCustomStack<hiprtGlobalStack, hiprtEmptyInstanceStack>
       closestTr(scene, ray, stack, instanceStack, hiprtFullRayMask,
       hiprtTraversalHintDefault, nullptr /* payload */, funcTable);
12 hiprtHit closestHit = closestTr.getNextHit();
```

There are three things in the code we have not explained yet: the instance stack, the ray mask, and the traversal hit. All three are not relevant for our use-case, but they could be useful in more complex scenarios. The instance stack is needed in the case of multi-level instancing. In our case, we use dummy empty stack that disables code for multi-level instancing and let compiler optimize the kernel more efficiently. We can optionally define a mask (32 bits) for each instance in the scene build input. This mask is tested against the ray mask (using the bitwise AND). In our case, as we have not specified these mask, they are assumed to be full such that the test returns always true. Last, the traversal hit can indicate from which distribution the ray has been generated (e.g., a shadow ray). However, in practice, it almost always best to use the default hint.

7.6 Trace Kernel Compilation

The last missing piece is the trace kernel compilation, which injects the HIPRT functionality to a user kernel. In principle, the trace kernel is nothing else than a regular HIP kernel. As HIPRT is an open-source project, the obvious way is to include the HIPRT headers directly in the kernel.

Another option is to use *bitcode* linking that allows to link pre-compiled HIPRT device code to the user kernel. In general, we can use for the compilation either by `HIPCC` or `HIPRTC` in runtime.

This approach works fine unless we use custom functions (e.g., the intersection function we defined above). Note that HIPRT supports besides the custom intersection functions also custom intersection filters that allow to filter out intersections during the traversal (e.g., alpha masking or filtering out self-intersections). The intersection filters are user callbacks similar to the custom intersection functions, but they can be used also for triangles compared to the custom intersection. The setup is practically the same as for the custom intersection functions. The custom functions make the compilation more complex as HIPRT needs to generate a function that dispatches custom functions based on the geometry type we set before. If we want to use `HIPCC`, we need to define the dispatch function manually.

Listing 27 shows how to compile the trace kernel in runtime using `hiprtBuildTraceKernels` function that builds the dispatch function out of the box. First, we create the function name-set and set the name of our custom intersection function. Then, we call `hiprtBuildTraceKernels` to compile the module, generating the dispatch function and injecting the HIPRT traversal code to the user kernel.

Listing 27. Trace kernel compilation via the HIPRT API.

```
hiprtFuncNameSet funcNameSet{};
funcNameSet.intersectFuncName = "intersectCustom";

hiprtApiFunction functionOut{};

std::string sourceCode = /* the source code of the HIP module */
std::string functionName = /* kernel function name */

hiprtBuildTraceKernels(
  hiprtCtx,
  1, /* the number of kernel functions */
  functionName.c_str(), /* the names of kernel functions */
  sourceCode.c_str(), /* the source code of the HIP module */
  "", /* path to the module source file (optional) */
  0, /* the number of the included headers (optional) */
  nullptr, /* sources of the included headers (optional) */
  nullptr, /* the names of included header (optional) */
  0, /* the number of compilations options (optional) */
  nullptr, /* compilation options (optional) */
  1, /* the number of geometry types */
  1, /* the number of ray types */
  &funcNameSet, /* custom function namesets (optional) */
  &functionOut, /* resulting HIP functions */
  nullptr, /* resulting HIP module (optional) */
  false /* cache compiled kernes (optional) */);
```

The signature of the function is similar to `hiprtcCreateProgram` in `HIPRTC`. There are a few optional arguments providing flexibility of the compilation. The mandatory arguments are as follows: the number of kernels to be compiled, the names of the kernels, and the source code of the module as a string. The compiled functions are in the form of `hiprtApiFunction`, which can be easily cast, for example, to `hipFunction_t` and launched using the standard HIP API.

Custom functions are passed as an array of `funcNameSet` structures. The layout of this array must correspond to the the layout of the function table we defined before. In our example, we have only one entry, and thus we pass just one `funcNameSet` object. For multiple functions, the layout

is a flattened 2D table with rows corresponding to ray types and columns to the geometry types stored in row-major order. Additional information about HIPRT can be found on GPUOpen[1].

8 TOPICS BEYOND THIS TECHNICAL REPORT

In this chapter, we presented an introduction to ray tracing with a focus on the GPU implementation in HIP. However, rendering is a vast area, and we touched only a very small fraction. In this section, we refer interested readers to additional topics for further exploration.

Rendering Algorithms We presented unidirectional path tracing with the next event estimation that is sufficient for most of the scenarios. Nevertheless, in complex settings, it might be difficult to reach the light source by tracing rays only from camera. For example, if the light source is enclosed by a glass sphere. In such cases, even the next event estimation fails because the light source is not directly visible. To tackle this problem, various bidirectional algorithms have been proposed. Bidirectional path tracing [4] casts rays from both the camera and the light sources, stochastically connecting these paths. Metropolis light transport [4] perturbs light paths using Markov chain Monte Carlo. Photon mapping [5] operates in two phases. In the first phase are cast photons from the light sources and stored in a spatial data structure. During the second phase, the camera rays are cast, collecting photons on the first diffuse surface. Photon mapping is particularly effective for rendering effects such as caustics.

Monte Carlo and Sampling An orthogonal approach to improving the efficiency of the light transport simulation is to sample light paths according to a distribution that matches the underlying light transport. This is called importance sampling (IS) in the context of Monte Carlo integration. The goal is to draw samples where the integrand has high values. The challenge is that besides the value we must also compute the corresponding probability density function. We showed a simple technique sampling according to the cosine term in this chapter. Multiple sampling techniques can be combined using multiple importance sampling (MIS) [4]. There is a way to improve rendering efficiency by changing the sampling points themselves, which is stratified sampling or Quasi-Monte Carlo sequences [13]. Such sequences are spatially distributed across the domain, and they outperform the convergence of Monte Carlo integration over random samples. Another widely-used technique is Russian roulette [14], which stochastically terminates paths based on their actual contribution, elegantly avoiding bias compared to a fixed depth while improving overall efficiency. We refer to *Advanced Global Illumination* by Dutré et al. [14] for more details about these methods.

Ray Tracing Since all these algorithms rely on ray tracing, efficiency can be further improved by optimizing the ray tracing, specifically the BVH. SBVH [15], a BVH construction method using spatial splits, is an approach that allows objects to be split to achieve tighter bounding boxes, albeit at the cost of higher memory consumption and more complex construction. H-PLOC [16] is a highly efficient construction algorithm, building BVHs of very good quality. SBVH and H-PLOC are implemented as high-quality and balanced options, respectively, in HIPRT.

Increasing Realism What is important alongside efficiency is the degree of realism. Material properties of surfaces have significant impact on realism, and thus modeling the appearance of real-world materials [17] is another active area of research. In a nutshell, the models describe how the light is reflected (or refracted) from different directions. Volumetric rendering [18] accounts for optical effects caused by, for instance, fog or smoke, bringing realism to another level. Spectral rendering [19] models wavelength-dependent effects such as dispersion and diffraction. We refer to *Physically-based Rendering* by Pharr et al. [7] that presents comprehensive overview of the techniques discussed so far.

[1] https://gpuopen.com/hiprt/

Real-time Rendering Ray tracing was originally designed for offline rendering, but thanks to advances in both hardware and software, ray tracing is gradually penetrating to real-time applications such as computer games. In this last paragraph, we discuss specifics for real-time scenarios. The major challenge in real-time ray tracing is that time budget is very limited to maintain real-time frame rates, allowing to trace only very few rays in each frame. Therefore, we need to use different tricks to get plausible results in several miliseconds. Denoising and upscaling [20] help mitigate noise caused by insufficient samples and enhance resolution, both of which nowadays rely predominantly on deep learning trained offline on extensive datasets. Radiance caching [21] stores radiance in a spatial data structure (or a neural network [22]), allowing to terminate light path earlier by approximating the remaining radiance by the cached values. ReSTIR [6; 23] is an algorithm for sampling a large number of light sources designed for real-time scenarios, resampling light candidates from neighboring pixels and form previous frames, exploiting spatiotemporal coherence.

REFERENCES

[1] T. Whitted, "An improved illumination model for shaded display," *Commun. ACM*, vol. 23, p. 343–349, June 1980.
[2] AMD, "Orochi." https://gpuopen.com/orochi/, 2020.
[3] J. T. Kajiya, "The rendering equation," *SIGGRAPH Comput. Graph.*, vol. 20, p. 143–150, Aug. 1986.
[4] E. Veach, *Robust monte carlo methods for light transport simulation.* PhD thesis, Stanford, CA, USA, 1998. AAI9837162.
[5] H. W. Jensen, *Realistic image synthesis using photon mapping.* USA: A. K. Peters, Ltd., 2001.
[6] B. Bitterli, C. Wyman, M. Pharr, P. Shirley, A. Lefohn, and W. Jarosz, "Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting," *ACM Trans. Graph.*, vol. 39, Aug. 2020.
[7] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: From Theory to Implementation (4rd ed.).* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 3rd ed., 2023.
[8] D. Meister, S. Ogaki, C. Benthin, M. J. Doyle, M. Guthe, and J. Bittner, "A survey on bounding volume hierarchies for ray tracing," *Computer Graphics Forum*, vol. 40, no. 2, pp. 683–712, 2021.
[9] I. Wald, "On Fast Construction of SAH-based Bounding Volume Hierarchies," in *Proceedings of Symposium on Interactive Ray Tracing*, pp. 33–40, 2007.
[10] B. Walter, K. Bala, M. Kulkarni, and K. Pingali, "Fast Agglomerative Clustering for Rendering," in *Proceedings of Symposium on Interactive Ray Tracing*, pp. 81–86, 2008.
[11] I. Wald, C. Benthin, and P. Slusallek, "Distributed Interactive Ray Tracing of Dynamic Scenes," in *Proceedings of Symposium on Parallel and Large-Data Visualization and Graphics*, pp. 77–86, 2003.
[12] D. Meister, P. Kulkarni, A. Vasishta, and T. Harada, "Hiprt: A ray tracing framework in hip," *Proc. ACM Comput. Graph. Interact. Tech.*, vol. 7, Aug. 2024.
[13] G. Singh, C. Öztireli, A. G. M. Ahmed, D. Coeurjolly, K. Subr, O. Deussen, V. Ostromoukhov, R. Ramamoorthi, and W. Jarosz, "Analysis of sample correlations for Monte Carlo rendering," *Computer Graphics Forum*, vol. 38, pp. 473–491, May 2019.
[14] P. Dutre, K. Bala, and P. Bekaert, *Advanced Global Illumination.* USA: A. K. Peters, Ltd., 2002.
[15] M. Stich, H. Friedrich, and A. Dietrich, "Spatial splits in bounding volume hierarchies," in *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, (New York, NY, USA), p. 7–13, Association for Computing Machinery, 2009.
[16] C. Benthin, D. Meister, J. Barczak, R. Mehalwal, J. Tsakok, and A. Kensler, "H-ploc: Hierarchical parallel locally-ordered clustering for bounding volume hierarchy construction," *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 7, pp. 30:1–30:14, Aug. 2024.
[17] D. Guarnera, G. Guarnera, A. Ghosh, C. Denk, and M. Glencross, "Brdf representation and acquisition," *Computer Graphics Forum*, vol. 35, no. 2, pp. 625–650, 2016.
[18] J. Novák, I. Georgiev, J. Hanika, and W. Jarosz, "Monte Carlo methods for volumetric light transport simulation," *Computer Graphics Forum (Proceedings of Eurographics - State of the Art Reports)*, vol. 37, May 2018.
[19] A. Weidlich, A. Forsythe, S. Dyer, T. Mansencal, J. Hanika, A. Wilkie, L. Emrose, and A. Langlands, "Spectral imaging in production: course notes siggraph 2021," in *ACM SIGGRAPH 2021 Courses*, SIGGRAPH '21, (New York, NY, USA), Association for Computing Machinery, 2021.
[20] P. Kazmierczyk, S. Kim, W. Uss, W. Kalinski, T. Galaj, M. Maciejewski, and R. Harihara, "Joint denoising and upscaling via multi-branch and multi-scale feature network," *Proc. ACM Comput. Graph. Interact. Tech.*, vol. 8, May 2025.

[21] J. Krivanek, P. Gautron, S. Pattanaik, and K. Bouatouch, "Radiance caching for efficient global illumination computation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, no. 5, pp. 550–561, 2005.

[22] T. Müller, F. Rousselle, J. Novák, and A. Keller, "Real-time neural radiance caching for path tracing," *ACM Trans. Graph.*, vol. 40, July 2021.

[23] C. Wyman, M. Kettunen, D. Lin, B. Bitterli, C. Yuksel, W. Jarosz, and P. Kozlowski, "A gentle introduction to restir path reuse in real-time," in *ACM SIGGRAPH 2023 Courses*, SIGGRAPH '23, (New York, NY, USA), Association for Computing Machinery, 2023.