CHAPTER 24

Real-Time Global Illumination with Photon Mapping

Niklas Smal and Maksim Aizenshtein
UL Benchmarks

ABSTRACT

Indirect lighting, also known as global illumination, is a crucial effect in photorealistic images. While there are a number of effective global illumination techniques based on precomputation that work well with static scenes, including global illumination for scenes with dynamic lighting and dynamic geometry remains a challenging problem. In this chapter, we describe a real-time global illumination algorithm based on photon mapping that evaluates several bounces of indirect lighting without any precomputed data in scenes with both dynamic lighting and fully dynamic geometry. We explain both the pre- and post-processing steps required to achieve dynamic high-quality illumination within the limits of a real-time frame budget.

24.1 INTRODUCTION

As the scope of what is possible with real-time graphics has grown with the advancing capabilities of graphics hardware, scenes have become increasingly complex and dynamic. However, most of the current real-time global illumination algorithms (e.g., light maps and light probes) do not work well with moving lights and geometry due to these methods' dependence on precomputed data.

In this chapter, we describe an approach based on an implementation of *photon mapping* [7], a Monte Carlo method that approximates lighting by first tracing paths of light-carrying photons in the scene to create a data structure that represents the indirect illumination and then using that structure to estimate indirect light at points being shaded. See Figure 24-1. Photon mapping has a number of useful properties, including that it is compatible with precomputed global illumination, provides a result with similar quality to current static techniques, can easily trade off quality and computation time, and requires no significant artist work. Our implementation of photon mapping is based on DirectX Raytracing (DXR) and gives high-quality global illumination with dynamic scenes. The overall structure of our approach is shown in Figure 24-2.

E. Haines, T. Akenine-Möller (eds.), *Ray Tracing Gems*, https://doi.org/10.1007/978-1-4842-4427-2_24

Figure 24-1. *Final result using our system.*

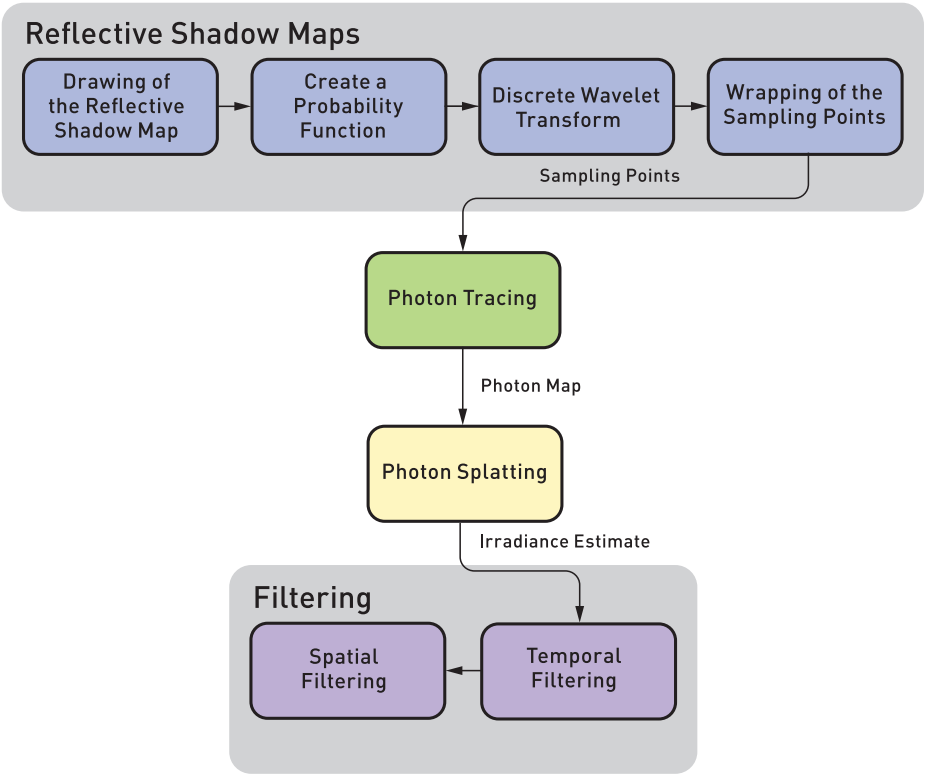


Figure 24-2. *The structure of the algorithm at the pass level. The first set of photons leaving the lights are taken care of using rasterization, producing a reflective shadow map. Points in these maps are sampled according to the power that their respective photons carry, and then ray tracing is used for subsequent photon bounces. To add indirect illumination to the final image, we splat photon contributions into the framebuffer using additive blending. Finally, temporal and spatial filtering are applied to improve image quality.*

Adapting photon mapping to real-time rendering on the GPU requires addressing a number of challenges. One is how to find nearby photons at points shaded in the scene so the photons can contribute indirect illumination to these locations. We found that an approach based on *splatting*, where each photon is rasterized into the image based on its contribution's extent, works well and is straightforward to implement.

Another challenge is that traditional photon mapping algorithms may not be able to reach the desired illumination quality within the computational constraints of real-time rendering. Therefore, we optimized the generation of photons using reflective shadow maps (RSMs) [2] to avoid tracing the first bounce of a ray from a light, replacing that step with rasterization. We are then able to apply importance sampling to the RSMs, choosing locations with high contributions more often to generate subsequent photon paths.

Finally, as is always the case when applying Monte Carlo techniques to real-time rendering, effective filtering is crucial to remove image artifacts due to low sample counts. To mitigate noise, we use temporal accumulation with an exponentially moving average and apply an edge-aware spatial filter.

24.2 PHOTON TRACING

While general ray tracing is necessary for following the paths of photons that have reflected from surfaces, it is possible to take advantage of the fact that all the photons leaving a single point light source have a common origin. In our implementation, the first segment of each photon path is handled via rasterization. For each emitter, we generate a *reflective shadow map* [2, 3], which is effectively a G-buffer of uniform samples of visible surfaces as seen from a light, where each pixel also stores the incident illumination. This basic approach was first introduced by McGuire and Luebke [10] nearly a decade ago, though they traced rays on the CPU at much lower performance and thus also had to transfer a significant amount of data between the CPU and the GPU—all of this fortunately no longer necessary with DXR.

After the initial intersection points are found with rasterization, photon paths continue by sampling the surface's BRDF and tracing rays. Photons are stored at all subsequent intersection points, to be used for reconstructing the illumination, as will be described in Section 24.3.

24.2.1 RSM-BASED FIRST BOUNCE

We start by selecting a total number of photons to emit from all light sources and then allocate these to lights proportional to each light's intensity. Hence, all photons initially carry roughly the same power. The RSM must contain all surface properties needed to generate rays for the initial bounce of the photons.

We choose to implement RSM generation as a separate pass that is executed after generating a traditional shadow map. Doing so allows us to make the resolution of the RSM map independent from the shadow map and keep its size constant, avoiding the need to allocate RSMs during runtime. As an optimization, it is possible to use the regular shadow map for depth culling. Without matching resolutions, this will give incorrect results for some pixels, but in our testing, we have not found it to cause visible artifacts.

After the RSMs are generated, we generate an importance map for sampling starting points for the first bounce where each RSM pixel is first given a weight based on the luminance of the product of the emitted power carried by the photon, including artist-controlled parameters such as directional falloff and the surface albedo. This weight value is directly related to the amount of power carried by photons that leave the surface.

This importance map is not normalized, which would be required for most sampling techniques. Rather than normalizing the map and generating sampling distributions, we instead apply a hierarchical sampling algorithm based on wavelet importance sampling, introduced by Clarberg et al. [1].

Wavelet importance sampling is a two-step algorithm. First, we apply the discrete Haar wavelet transform to the probability map, effectively generating a pyramid representation of the image. Second, we reconstruct the signal for each sample location in a low-discrepancy sequence and warp the sampling positions based on the scaling coefficient of each iteration in a wavelet transformation. This warping scheme is illustrated in Figure 24-3. See also Chapter 16, "Sampling Transformations Zoo," for more information about it.

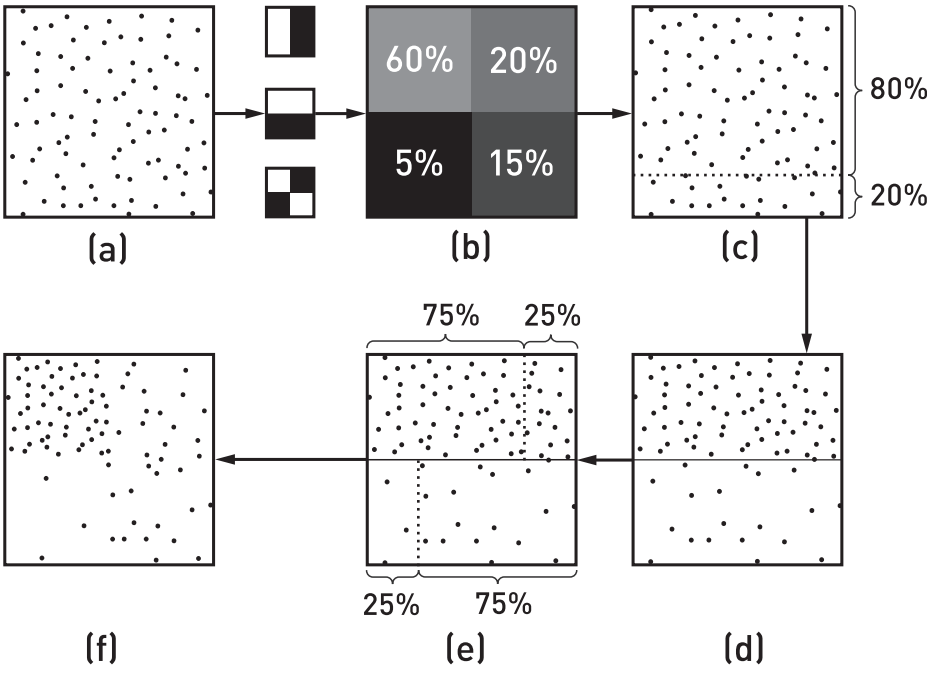


Figure 24-3. *Warping a set of sampling positions by an iteration of the wavelet transformation. (a) The initial sampling positions are (c–d) first warped horizontally and (e–f) then vertically using (b) the ratios of the scaling coefficients in the active quad. (Illustration after Clarberg et al. [1].)*

The wavelet transformation must be applied across the entire image pyramid, at halved resolutions at each step, ending at 2 × 2 resolution. Because launching individual compute shader passes for such small dimensions is inefficient, we implement a separate compute shader pass for the final levels that uses memory similarly to a standard reduction implementation.

Importance sampling transforms the low-discrepancy samples into sample positions in the RSM with associated probabilities. In turn, a direction for an outgoing ray is found using importance sampling. Sampled rays are represented using the format presented in Table 24-1. Because each sample is independent from the other samples, there is no need for synchronization between sample points, except for an atomic counter to allocate a location in the output buffer. However, we must generate the seeds for the random number generator at this stage using the sampling index instead of later in photon tracing using the sample buffer location; doing so keeps photon paths deterministic between frames.

Table 24-1. *Format for sampled points.*

Property	Format
Position	`float3`
Direction	`3 × float16`
Power	`uint`—Shared Exponent Packing
Seed	`uint`
Padding	`uint`

By using importance sampling to select the pixels in the RSM from which photons are traced, we are able to select the pixels whose photons carry more power more frequently. This in turn leads to less variation in photon power. Another advantage of RSMs is that they make it easy to trace multiple photon paths from an RSM point, selecting a different direction for each one. Doing so is useful when the desired photon count becomes high compared to the resolution of the RSM.

24.2.2 FOLLOWING PHOTON PATHS

Starting with the sampled RSM points and then at each subsequent photon/surface intersection, we generate an outgoing direction ω using importance sampling with a sampling distribution $p(\omega)$ that is similar to the surface's BRDF. For example, we use a cosine-weighted distribution for diffuse surfaces, and we sample the visible microfacet distribution for microfacet BRDFs [5].

Before tracing the reflected photon, however, we apply *Russian roulette*, randomly terminating the photon based on the ratio between the BRDF times ($\omega \cdot \omega_g$) and the sampled direction probability. Photons that survive this test have their contribution adjusted accordingly so that the end result is correct. In this way, when a ray encounters a surface that reflects little light, fewer photons continue than if the surface reflects most of the incident light. Just like allocating photons to lights based on their emitted power, this also improves results by ensuring that all live photons have roughly the same contribution.

Since the power of a photon has multiple channels (in the RGB color model), the Russian roulette test can be modified so that it is done once, instead of per channel. We choose to handle this with the solution described by Jensen [7], setting the termination probability as

$$q = \frac{\max\left(\rho_r \Phi_{i,r}, \max\left(\rho_g \Phi_{i,g}, \rho_b \Phi_{i,b}\right)\right)}{\max\left(\Phi_{i,r}, \max\left(\Phi_{i,g}, \Phi_{i,b}\right)\right)}, \tag{1}$$

where q is the scalar termination probability, Φ_i is the incoming power of the photon, and ρ is the ratio between the BRDF times ($\omega \cdot \omega_g$) and the scattering direction probability density function (PDF). The outgoing photon power is then $\Phi_i \frac{\rho}{q}$ with component-wise multiplication.

Instead of using the same random samples for every frame, we are careful to use a new random seed each time. This causes the paths for the photons traced to vary for each frame, thus providing a different sample set and leading to accumulation of the larger sample set over multiple frames.

Photons are stored in an array where entries are allocated by atomically incrementing a global counter. Since our purpose is to calculate only indirect lighting, we do not store a photon for the initial photon/surface intersection in the RSM, as it represents direct illumination, which is better handled using other techniques (e.g., shadow maps or tracing shadow rays). We also do not store photons at surfaces with normals facing away from the camera or photons that are located outside of the camera frustum—both types do not contribute to the final image and are best culled before splatting. Note that our frustum culling considers photons only as points and ignores their splat radius. Thus, some photons at the edge of the frustum that actually would contribute to the radiance estimate are incorrectly culled. This issue could possibly be addressed by expanding the camera frustum used for the culling. However, this error does not seem to cause any significant visual artifacts when the kernel size in screen space is sufficiently small.

The representation of each photon is 32 bytes and is presented in Table 24-2.

Table 24-2. *Representation of a photon.*

Property	Format
Position	`float3`
Power	uint—Shared Exponent Packing
Normal	2 × float16—Stereographic Packing
Light Direction	2 × float16—Stereographic Packing
Ray Length	`float`
Sign Bits for Normal/Direction and Padding	`uint`

24.2.3 DXR IMPLEMENTATION

Implementing photon tracing using DXR is fairly simple: a ray generation shader is invoked for all the RSM points that have been sampled, using each as a starting point for subsequent photon rays. It is then responsible for tracing subsequent rays until either a maximum number of bounces is reached or the path is terminated by Russian roulette.

Two optimizations are important for performance. The first is minimizing the size of the ray payload. We used a 32-byte ray payload, encoding the ray direction using 16-bit `float16` values and the RGB photon power as a 32-bit `rgb9e5` value. Other fields in the payload store the state of the pseudo-random number generator, the length of the ray, and the number of bounces.

The second key optimization is to move the logic for sampling new ray directions and applying Russian roulette to the closest-hit shader. Doing so significantly improves performance by reducing register pressure. Together, we have the following for the ray generation shader:

```
struct Payload
{
    // Next ray direction, last element is padding
    half4 direction;
    // RNG state
    uint2 random;
    // Packed photon power
    uint power;
    // Ray length
    float t;
    // Bounce count
    uint bounce;
};

[shader("raygeneration")]
void rayGen()
{
    Payload p;
    RayDesc ray;

    // First, we read the initial sample from the RSM.
    ReadRSMSamplePosition(p);

    // We check if bounces continue by the bounce count
    // and ray length (zero for terminated trace or miss).
    while (p.bounce < MAX_BOUNCE_COUNT && p.t != 0)
    {
        // We get the ray origin and direction for the state.
        ray.Origin = get_hit_position_in_world(p, ray);
        ray.Direction = p.direction.xyz;

```

```
        TraceRay(gRtScene, RAY_FLAG_FORCE_OPAQUE, 0xFF, 0,1,0, ray, p);
        p.bounce++;
    }
}
```

The closest-hit shader unpacks the required values from the ray payload and then determines which ray to trace next. The `validate_and_add_photon()` function, to be defined shortly, stores the photon in the array of saved photons, if it is potentially visible to the camera.

```
[shader("closesthit")]
void closestHitShader(inout Payload p : SV_RayPayload,
    in IntersectionAttributes attribs : SV_IntersectionAttributes)
{
    // Load surface attributes for the hit.
    surface_attributes surface = LoadSurface(attribs);

    float3 ray_direction = WorldRayDirection();
    float3 hit_pos = WorldRayOrigin() + ray_direction * t;
    float3 incoming_power = from_rbge5999(p.power);
    float3 outgoing_power = .0f;

    RandomStruct r;
    r.seed = p.random.x;
    r.key = p.random.y;

    // Russian roulette check
    float3 outgoing_direction = .0f;
    float3 store_power = .0f;
    bool keep_going = russian_roulette(incoming_power, ray_direction,
        surface, r, outgoing_power, out_going_direction, store_power);

    repack_the_state_to_payload(r.key, outgoing_power,
    outgoing_direction, keep_going);

    validate_and_add_photon(surface, hit_pos, store_power,
            ray_direction, t);
}
```

Finally, as described earlier in Section 24.2, the photons that are stored are added to a linear buffer, using atomic operations to allocate entries.

```
void validate_and_add_photon(Surface_attributes surface,
    float3 position_in_world, float3 power,
    float3 incoming_direction, float t)
{
    if (is_in_camera_frustum(position) &&
        is_normal_direction_to_camera(surface.normal))
```

```
 7     {
 8         uint tile_index =
 9             get_tile_index_in_flattened_buffer(position_in_world);
10         uint photon_index;
11         // Offset in the photon buffer and the indirect argument
12         DrawArgumentBuffer.InterlockedAdd(4, 1, photon_index);
13         // Photon is packed and stored with correct offset.
14         add_photon_to_buffer(position_in_world, power, surface.normal,
15             power, incoming_direction, photon_index, t);
16         // Tile-based photon density estimation
17         DensityEstimationBuffer.InterlockedAdd(tile_i * 4, 1);
18     }
19 }
```

24.3 SCREEN-SPACE IRRADIANCE ESTIMATION

Given the array of photons, the next task is to use them to reconstruct indirect illumination in the image. Each photon has a kernel associated with it that represents the extent of the scene (and thus, the image) to which it possibly contributes. The task is to accumulate each photon's contribution at each pixel.

Two general approaches have been applied to this problem: *gathering* and *scattering*. Gathering is essentially a loop over pixels, where at each pixel nearby photons are found using a spatial data structure. Scattering is essentially a loop over photons, where each photon contributes to the pixels that it overlaps. See Mara et al. [9] for a comprehensive overview of both real-time gathering and scattering techniques. Given highly efficient ray tracing on modern GPUs to generate photon maps, it is also important that reconstruction be efficient. Our implementation is based on scattering and we take advantage of rasterization hardware to efficiently draw the splatting kernels. Results are accumulated using blending.

We use photons to reconstruct *irradiance*, which is the cosine-weighted distribution of light arriving at a point. We then approximate the light reflected from a surface by the product of the photon's irradiance and the surface's BRDF using a mean incoming direction. In doing so, we discard the directional distribution of indirect illumination and avoid a costly evaluation of the reflection model for every photon that influences a point's shading. This gives the correct result for diffuse surfaces, but it introduces error as surfaces become more glossy and as the distribution of indirect lighting becomes more irregular. In practice, we have not seen objectionable errors from this approximation.

24.3.1 DEFINING THE SPLATTING KERNEL

Selecting a good kernel size for each photon is important: if the kernels are too wide, the lighting will be excessively blurry, and if they are too narrow, it will be blotchy. It is particularly important to avoid too-wide kernels because a wider kernel makes a photon cover more pixels and thus leads to more rasterization, shading, and blending work for the photon. Incorrect kernel selection for photon mapping can cause several types of biases and errors [14]; minimization of these has been the focus of a substantial amount of research.

In our approach, we start with a spherical kernel and then apply a number of modifications to it in order to minimize various types of error. These modifications can be categorized into two main types: uniform scaling and modification of the kernel's shape.

24.3.1.1 UNIFORM SCALING OF THE KERNEL

Uniform scaling of the kernel is a product of two terms, the first one based on the ray length and the second on an estimation of the photon density distribution.

Ray Length We scale the kernel according to the ray length using linear interpolation to a constant maximum length. This method is an approximation of the ray differential and can be interpreted as treating the photon as traveling along a cone instead of a ray and factoring in the growth of the cone base as its height increases. Also, we can assume lower photon densities as the ray length increases, since it is probable that photons scatter to a larger world-space volume. Thus, we want a relatively wide kernel in that case. The scaling factor is

$$s_l = \min\left(\frac{l}{l_{\max}}, 1\right), \tag{2}$$

where l is the ray length and $l_{\max}$ is a constant defining the maximum ray length. However, $l_{\max}$ is not required to be the maximum length of the rays cast during photon tracing but instead the length that we consider to be the maximum height of the cone. This constant should be related to the overall scale of the scene and can be derived from its bounding box.

Photon Density We would like to further scale each photon's kernel based on the local photon density around it: the more photons that are nearby, the smaller the kernel can (and should) be. The challenge is efficiently determining how many photons are near each one. We apply the simple approximation of maintaining a counter for each screen-space tile. When a photon is deposited in a tile, the counter

is atomically incremented. This is obviously a crude approximation of the density function, but it seems to produce fairly good results.

We then implement density-based scaling as a function of the area of the tile in view space:

$$a_{\text{view}} = z_{\text{view}}^2 \frac{\tan(\alpha_x/2)\tan(\alpha_y/2)t_x t_y}{r_x r_y}, \tag{3}$$

where α_x and α_y are the apertures of the camera frustum, z_{view} is the distance from the camera, t_x and t_y are the tile dimensions in pixels, and r_x and r_y represent the image's resolution. In most cases a tile does not have a uniform depth, so we use the depth of the photon position. Most of this arithmetic can be precalculated and replaced with a camera constant:

$$a_{\text{view}} = z_{\text{view}}^2 c_{\text{tile}}. \tag{4}$$

Thus, scaling the circular kernel to have the same area in the view space as the tile can be calculated as

$$a_{\text{view}} = \pi r^2 n_p, \quad r = \sqrt{\frac{z_{\text{view}}^2 c_{\text{tile}}}{\pi n_p}}, \tag{5}$$

where n_p is the number of photons in the tile. This value is clamped to remove any extreme cases and then multiplied by the constant n_{tile}, which is equal to the number of photons that we expect to contribute to each pixel:

$$s_d = \text{clamp}(r, r_{\min}, r_{\max}) n_{\text{tile}}. \tag{6}$$

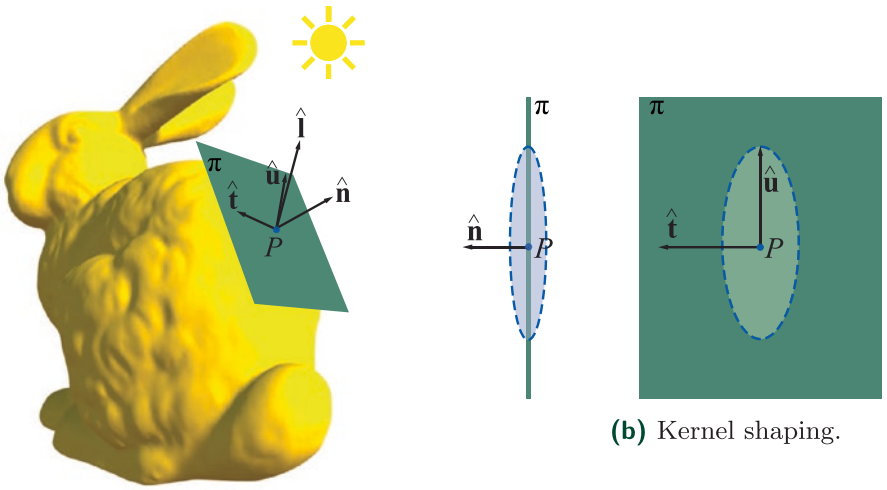
The HLSL implementation of these equations is straightforward:

```
1 float uniform_scaling(float3 pp_in_view, float ray_length)
2 {
3     // Tile-based culling as photon density estimation
4     int n_p = load_number_of_photons_in_tile(pp_in_view);
5     float r = .1f;
6
7     if (layers > .0f)
8     {
9         // Equation 5
10        float a_view = pp_in_view.z * pp_in_view.z * TileAreaConstant;
11        r = sqrt(a_view / (PI * n_p));
12    }
```

```
13      // Equation 6
14      float s_d = clamp(r, DYNAMIC_KERNEL_SCALE_MIN,
15          DYNAMIC_KERNEL_SCALE_MAX) * n_tile;
16
17      // Equation 2
18      float s_l = clamp(ray_length / MAX_RAY_LENGTH, .1f, 1.0f);
19      return s_d * s_l;
20  }
```

24.3.1.2 ADJUSTING THE KERNEL'S SHAPE

We can further improve the reconstructed result by adjusting the kernel's shape. We consider two factors. First, we decrease the radius of the kernel in the direction of the normal of the surface that the photon intersected. Second, we scale the kernel in the direction of the light in order to model the projected area that it covers on the surface. This results in the kernel being a tri-axial ellipsoid, which has one axis, **n**, that has the direction ω_g of the normal. The other two axes are placed on a tangent plane defined by the photon normal, called the *kernel plane*. The first of the two, **u**, has the direction of ω_i projected onto the kernel plane, while the second, **t**, is orthogonal to it and in the same plane. This vector basis is illustrated in Figure 24-4.



(a) Base vectors.

(b) Kernel shaping.

Figure 24-4. *Left: the base vectors for the kernel space: ω_g is aligned to the photon normal $\hat{\boldsymbol{n}}$, which also defines the kernel plane π. Two other basis vectors lie in π such that $\hat{\boldsymbol{u}}$ is the projection of light direction ω_i on to the kernel plane and $\hat{\boldsymbol{t}}$ is orthogonal to $\hat{\boldsymbol{u}}$. Right: the kernel's shape is modified by scaling along those vectors.*

The magnitude of **n** is $s_n s_l s_d$, where s_n is a constant that compresses the kernel along the normal so that it is closer to the surface. This is a common approach: it was done by Jensen [7] for gathering with a varying gathering radius and by

McGuire and Luebke [10] for their splatting kernel. Compared to a spherical kernel, this provides a better approximation of the surface. However, if the kernel is compressed too much, the distribution on objects with complex shapes or significant surface curvatures becomes inaccurate, as the kernel disregards samples farther away from its plane. This can be compensated for by making the magnitude be a function of the surface curvature, but in our implementation this factor is constant.

The magnitude of **u** is $s_u s_l s_d$, where s_u is defined as a function of the cosine of the angle between the hit normal and the light direction:

$$s_u = \min\left(\frac{1}{\omega_g \cdot \omega_i},\ s_{\max}\right), \tag{7}$$

where $s_{\max}$ is a constant defining the maximum scaling factor. Otherwise, the magnitude would approach infinity as the angle between ω_g and ω_i decreases to zero. Intuition for this equation originates in ray differentials and the cone representation of the photon: as the incoming direction of the photon becomes orthogonal to the normal direction of the surface, the area of the base of the cone that is projected onto the kernel plane increases.

Finally, the magnitude of t is $s_l s_d$.

The following code shows an implementation of the shape modification:

```
kernel_output kernel_modification_for_vertex_position(float3 vertex,
    float3 n, float3 light, float3 pp_in_view, float ray_length)
{
    kernel_output o;
    float scaling_uniform = uniform_scaling(pp_in_view, ray_length);

    float3 l = normalize(light);
    float3 cos_alpha = dot(n, vertex);
    float3 projected_v_to_n = cos_alpha * n;
    float cos_theta = saturate(dot(n, l));
    float3 projected_l_to_n = cos_theta * n;

    float3 u = normalize(l - projected_l_to_n);

    // Equation 7
    o.light_shaping_scale = min(1.0f/cos_theta, MAX_SCALING_CONSTANT);

    float3 projected_v_to_u = dot(u, vertex) * u;
    float3 projected_v_to_t = vertex - projected_v_to_u;
    projected_v_to_t -= dot(projected_v_to_t, n) * n;

```

```
22      // Equation 8
23      float3 scaled_u = projected_v_to_u * light_shaping_scale *
24          scaling_Uniform;
25      float3 scaled_t = projected_v_to_t * scaling_uniform;
26      o.vertex_position = scaled_u + scaled_t +
27           (KernelCompress * projected_v_to_n);
28
29      o.ellipse_area = PI * o.scaling_uniform * o.scaling_uniform *
30           o.light_shaping_scale;
31
32      return o;
33  }
```

24.3.2 PHOTON SPLATTING

We splat photons using an instanced indirect draw of an icosahedron as an approximation to a sphere. (The indirect arguments for the draw call are set using an atomic counter in the `validate_and_add_photon()` function.) To apply the kernel shape introduced in the previous section, we transform the vertices in the vertex shader accordingly. Since the original kernel is a sphere, we can assume the coordinate frame of the kernel's object space to be the coordinate frame of the world space, which results in vertex positions

$$\mathbf{v}_{\text{kernel}} = \begin{pmatrix} \mathbf{n} & \mathbf{u} & \mathbf{t} \end{pmatrix} \begin{pmatrix} \hat{\mathbf{n}}^{\mathsf{T}} \\ \hat{\mathbf{u}}^{\mathsf{T}} \\ \hat{\mathbf{t}}^{\mathsf{T}} \end{pmatrix} \mathbf{v}. \tag{8}$$

We keep the pixel shader for our splatting kernel as simple as possible, as it can easily become a performance bottleneck. Its main task is a depth check to ensure that the G-buffer surface for which we are calculating radiance is within the kernel. The depth check is done as a clipping operation for the world-space distance between the surface and the kernel plane against a constant value scaled by the kernel compression constant. After the depth check, we apply the kernel to the splatting result:

$$E_i = \frac{\Phi}{a}, \tag{9}$$

where a is the area of the ellipse, $a = \pi\|\mathbf{u}\|\|\mathbf{t}\| = \pi(s_l s_d)(s_l s_d s_u)$. It is worth noting that irradiance here is scaled by the cosine term and thus implicitly includes information from the geometric normals.

For accumulation of irradiance, we use a half-precision floating-point format (per channel) in order to avoid numerical issues with lower-bit formats. Furthermore, we accumulate the average light direction as a weighted sum with half-precision floats. The motivation for also storing the direction is discussed in Section 24.4.3.

The following code implements splatting. It uses the two functions defined previously to adjust the kernel's shape.

```
void VS(
    float3 Position : SV_Position,
    uint instanceID : SV_InstanceID,
    out vs_to_ps Output)
{
    unpacked_photon up = unpack_photon(PhotonBuffer[instanceID]);
    float3 photon_position = up.position;
    float3 photon_position_in_view = mul(WorldToViewMatrix,
    float4(photon_position, 1)).xyz;
    kernel_output o = kernel_modification_for_vertex_position(Position,
    up.normal, -up.direction, photon_position_in_view, up.ray_length);

    float3 p = pp + o.vertex_position;

    Output.Position = mul(WorldToViewClipMatrix, float4(p, 1));
    Output.Power = up.power / o.ellipse_area;
    Output.Direction = -up.direction;
}

[earlydepthstencil]
void PS(
vs_to_ps Input,
out float4 OutputColorXYZAndDirectionX : SV_Target,
out float2 OutputDirectionYZ : SV_Target1)
{
    float depth = DepthTexture[Input.Position.xy];
    float gbuffer_linear_depth = LinearDepth(ViewConstants, depth);
    float kernel_linear_depth = LinearDepth(ViewConstants,
        Input.Position.z);
    float d = abs(gbuffer_linear_depth - kernel_linear_depth);

    clip(d > (KernelCompress * MAX_DEPTH) ? -1 : 1);

    float3 power = Input.Power;
    float total_power = dot(power.xyz, float3(1.0f, 1.0f, 1.0f));
    float3 weighted_direction = total_power * Input.Direction;

    OutputColorXYZAndDirectionX = float4(power, weighted_direction.x);
    OutputDirectionYZ = weighted_direction.yz;
}
```

As mentioned before, we use additive blending to accumulate the contributions of photons. Modern graphics APIs guarantee that pixel blending occurs in submission order, though we do not need this property here. As an alternative, we tried using raster order views but found that these were slower than blending. However,

using floating-point atomic intrinsics, which are available on NVIDIA GPUs as an extension, did result in improved performance in situations when many photons overlap in screen space (a common scenario for caustics).

24.3.2.1 OPTIMIZING SPLATTING USING REDUCED RESOLUTION

Splatting can be an expensive process, which is especially the case when rendering high-resolution images. We found that reducing image resolution to half of the native rendering resolution did not cause a noticeable decrease in visual quality for the final result and gave a significant performance benefit. Using lower resolution does require a change to the depth clipping in the pixel shader to eliminate irradiance bleeding between surfaces: the half-resolution depth stencil used for stencil drawing should be downscaled using the closest pixel to the camera, but the depth used in pixel shader clipping should be downscaled using the farthest pixel from the camera. Hence, we draw the splatting kernel for only those pixels that are entirely within the full-resolution kernel. This causes jagged edges in the splatting result, but they are removed by the filtering.

24.4 FILTERING

As typical for real-time Monte Carlo rendering methods, it is necessary to apply image filtering algorithms to compensate for the low sample count. Although there have been significant advances in denoising in recent years, the noise caused by photon distribution kernels is quite different from the high-frequency noise that path tracing exhibits and that has been the main focus of denoising efforts. Thus, a different solution is required.

We use both temporal and spatial accumulation of samples with geometry-based edge-stopping functions. Our approach is based on previous work by Dammertz et al. [4] and Schied et al. [13], with our implementation using an edge-avoiding À-Trous wavelet transform for spatial filtering. Because indirect lighting is generally low frequency, we considered filtering at a lower resolution to decrease the computation cost, but we encountered artifacts due to G-buffer discrepancies and so reverted to filtering at the final resolution.

Both our temporal and spatial filtering algorithms use *edge-stopping functions* based on the difference in depth between two pixels and the difference in their surface normals. These functions, based on those of Schied et al. [13], attempt to prevent filtering across geometric boundaries by generating weights based on the

surface attributes of two different pixels p and q. The depth difference weight w_z is defined by

$$w_z(P,Q) = \exp\left(-\frac{|z(P)-z(Q)|}{\sigma_z|\nabla\mathbf{z}(P)(P-Q)|+\varepsilon}\right), \tag{10}$$

where $z(P)$ is the screen-space depth at a pixel location P and $\nabla\mathbf{z}(P)$ is the depth gradient vector. After experimentation $\sigma_z = 1$ was found to work well.

Next, the normal difference weight w_n accounts for the difference of the surface normals:

$$w_n(P,Q) = \max\left(0, \hat{\mathbf{n}}(P)\cdot\hat{\mathbf{n}}(Q)\right)^{\sigma_n}, \tag{11}$$

where we found $\sigma_n = 32$ to work well.

24.4.1 TEMPORAL FILTERING

Temporal filtering improves image quality by accumulating values from previous frames. It is implemented by reprojecting the previous frame's filtered irradiance values using velocity vectors and then at every pixel p computing an exponentially moving average between the previous frame's reprojected filtered irradiance value $\tilde{E}_{i-1}(P)$ and the irradiance value $E_i(P)$ computed using splatting, giving a temporally filtered irradiance $\tilde{E}_i(P)$:

$$\tilde{E}_i(P) = (1-\alpha)E_i(P) + \alpha\tilde{E}_{i-1}(P). \tag{12}$$

This is Karis's temporal antialiasing (TAA) approach [8] applied to irradiance.

Using a constant value for α would cause severe ghosting artifacts, as the temporal filtering would not account for disocclusions, moving geometry, or moving lights. However, because the irradiance values E_i at a pixel can vary significantly between frames, color-space clipping methods used in TAA are not well suited for them. Therefore, we rely on geometry-based methods and define α using the edge-stopping functions as

$$\alpha = 0.95\, w_z(P,Q)\, w_n(P,Q), \tag{13}$$

where P is a current pixel sample and Q is the projected sample from the previous frame. To evaluate the weight functions, it is necessary to maintain the normal and depth data from the G-buffer of the previous frame. If the resolution of the splatting target is lower than the filtering target, we upscale the splatting result at the beginning of the temporal filtering pass with bilinear sampling.

24.4.2 SPATIAL FILTERING

The edge-avoiding Á-Trous wavelet transform is a multi-pass filtering algorithm with an increasing kernel footprint Ω_i at each step i. This is illustrated in one dimension in Figure 24-5. Note that the spacing of filter taps doubles at each stage and that intermediate samples between filter taps are just ignored. Thus, the filter can have a large spatial extent without an excessive growth in the amount of computation required. The algorithm is particularly well suited to GPU implementation, where group shared memory can be used to efficiently share surface attributes across different pixels evaluating the kernel.

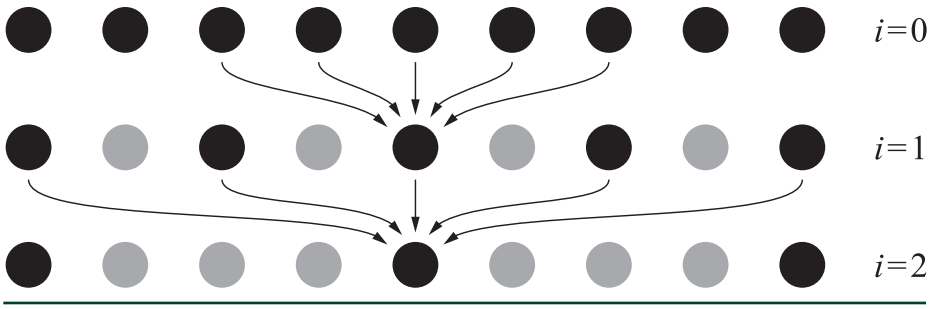


Figure 24-5. *Three iterations of the one-dimensional stationary wavelet transform that forms the basis of the Á-Trous approach. Arrows show the nonzero elements of the previous result contributing to the current element, while gray dots indicate zero elements. (Illustration after Dammertz et al. [4].)*

Our implementation follows Dammertz et al. [4] and Schied et al. [13] in which we realize each iteration as a 5 × 5 cross bilateral filter. Contributing samples are weighted by a function $w(P, Q)$, where P is the current pixel and Q is the contributing sample pixel within the filter. The first iteration uses the temporally filtered irradiance values

$$s_0(P) = \frac{\sum_{Q \in \Omega_0} h(Q) w(P, Q) \tilde{E}_i(Q)}{\sum_{Q \in \Omega_0} h(Q) w(P, Q)}, \tag{14}$$

and then each following level filters the previous one:

$$s_{i+1}(P) = \frac{\sum_{Q \in \Omega_i} h(Q) w(P, Q) s_i(Q)}{\sum_{Q \in \Omega_i} h(Q) w(P, Q)}, \tag{15}$$

where $h(Q) = \left(\frac{1}{8}, \frac{1}{4}, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}\right)$ is the filter kernel and $w(P, Q) = w_z(P, Q) w_n(P, Q)$.

24.4.2.1 VARIANCE CLIPPING OF DETAIL COEFFICIENTS

To avoid blurring excessively, it is important to adapt the image filtering based on the accuracy of the image contents. For example, Schied et al. [13] use an estimate of variance as a part of their weight function. That works well for high-frequency noise but is unsuitable for the low-frequency noise from photon mapping. Therefore, we have developed a new filtering algorithm based on variance clipping of the differences between each stage of the Á-Trous transform.

The *stationary wavelet transform* (SWT) was originally introduced to combat one of the shortcomings of the discrete wavelet transform, that the transformation is not shift-invariant. This problem was solved by saving detail coefficients per pixel for each iteration. The detail coefficients can be defined by

$$d_i = s_{i+1} - s_i. \tag{16}$$

This makes the SWT inherently redundant. If we consider how to reconstruct the original signal, we have

$$s_0 = s_n - \sum_{i=0}^{n-1} d_i, \tag{17}$$

where n is the number of iterations. As we can see, to reconstruct the original signal we need only the sum of the detail coefficients. Doing so allows us to reduce the amount of required memory to two textures, each with the resolution of the original image. Nevertheless, this just leaves us at the same point where we started—the original unfiltered image.

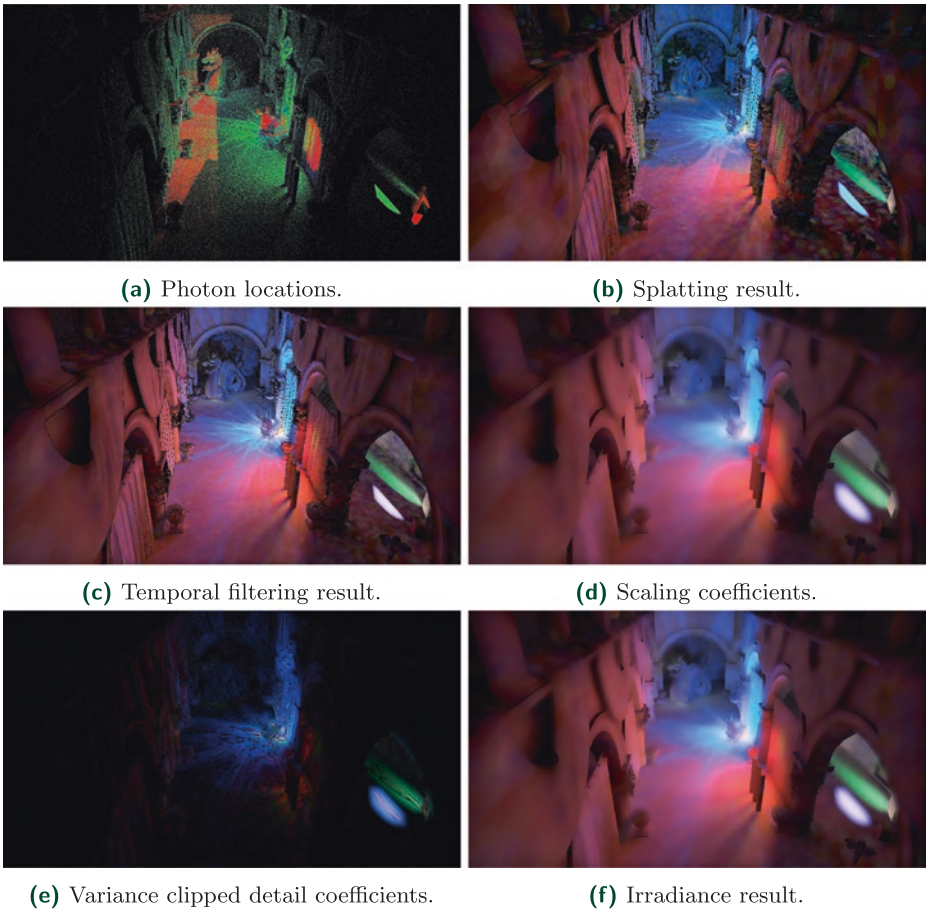
However, we can apply variance clipping [12] to each of the detail coefficients before we add them in to the sum. This approach works well here, unlike with the unfiltered frame irradiance values E_i, because we are starting with temporally filtered values. We compute color-space boundaries (denoted by b_i) based on the variance of irradiance within the spatial kernel. In turn, these boundaries are used for clipping the detail coefficients, and we compute a final filtered irradiance value as

$$E_{\text{final}} = s_n - \sum_{i=0}^{n-1} \text{clamp}\left(d_i, -b_i, b_i\right). \tag{18}$$

Finally, we apply the filtered irradiance to the surfaces. As described earlier, we ignore the directional distribution of indirect lighting. Instead, we use the mean direction as the incoming light parameter to evaluate the BRDF. The irradiance is multiplied by the BRDF value retrieved:

$$L_{\text{final}} = E_{\text{final}}\, f_{\text{BRDF}}. \tag{19}$$

Figure 24-6 illustrates the various passes of our approach.



(a) Photon locations. **(b)** Splatting result.

(c) Temporal filtering result. **(d)** Scaling coefficients.

(e) Variance clipped detail coefficients. **(f)** Irradiance result.

Figure 24-6. *Results for different passes of the algorithm using the Sponza scene with three bounces of indirect light, four light sources, three million initial photons, and four spatial filtering iterations. (a) The colors red, green, and blue correspond to the number of the bounce. Notice the accumulation of different sample subsets from (c) stratified sampling in temporal filtering result compared to (b) the splatting result. Also, (e) the effect of the variance clipping of detail coefficients is clearly visible as (f) the irradiance result retains much of detail that is lost when (d) only scaling coefficients are used.*

24.4.3 INCORPORATING THE EFFECT OF SHADING NORMALS

Photon mapping includes the directional information as an implicit part of the irradiance calculation, because surfaces with their geometry normals pointing toward the incoming light direction have a higher probability of being hit by photons. However, this process does not capture the detail provided by material attributes, such as normal maps. This is a commonly known issue

with precalculated global illumination methods, and there have been several approaches to solve it [11]. To achieve comparable illumination quality, we must also take this factor into consideration with photon mapping.

We developed a solution inspired by Heitz et al. [6]: we filter the light direction ω_i as a separate term. Then, when computing the irradiance, we effectively remove the original cosine term from the dot product of this direction ω_i with the geometric normal ω_{ng} and replace it with the dot product of ω_i and the shading normal ω_{ns}. This changes Equation 19 to

$$L_{\text{final}} = E_{\text{final}} f_{\text{BRDF}}\left(\omega_i \cdot \omega_{ns}\right) \min\left(\frac{1}{\left(\omega_i \cdot \omega_{ng}\right) + \varepsilon}, s_{\max}\right), \tag{20}$$

where ω_i is the weighted average of the light directions and $s_{\max}$ is the maximum scaling factor used in Section 24.3.1.2.

Accounting for the surface normal in this way comes with a performance cost, as it requires an additional blending target for the splatting, along with additional input and output for each filtering step. However, it allows us to apply the information from the normal maps without reading the G-buffer normal when computing irradiance.

Filtering the BRDF instead of just the light direction would achieve more accurate results for specular surfaces. However, doing so would require evaluating the BRDF during irradiance estimation and thus reading the material attributes. This would come with a significant performance cost when implemented with splatting, as the G-buffer would have to be read for each pixel shader invocation. A compute shader–based gathering approach could avoid this problem by loading the material attributes only once, though it would still pay the computational cost of evaluating the BRDF.

24.5 RESULTS

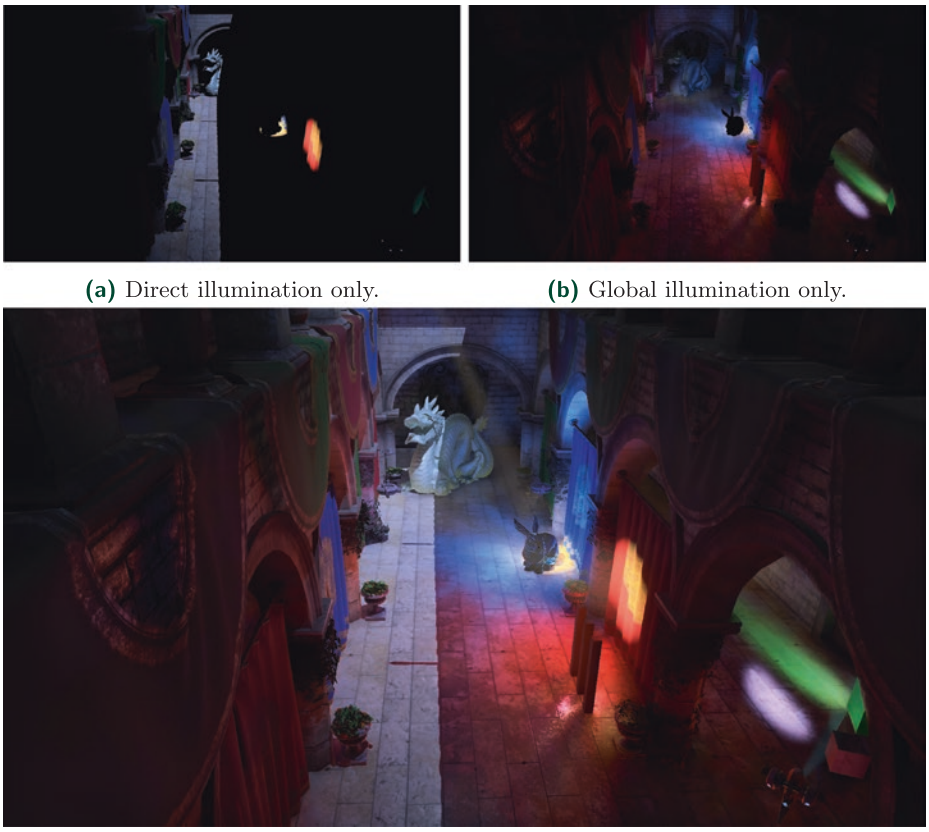
We evaluated our implementation with three scenes: Conference Room (shown in Figure 24-7), Sponza (Figure 24-8), and 3DMark Port Royal (Figure 24-9). Conference Room has a single light source, Sponza has four, and Port Royal has one spotlight from a drone and another pointing toward the camera. The rendering of the Port Royal scene includes an artistic multiplier to the photon power in order to intensify the indirect lighting effect.

(a) Direct illumination only.

(b) Global illumination only.

(c) Final result.

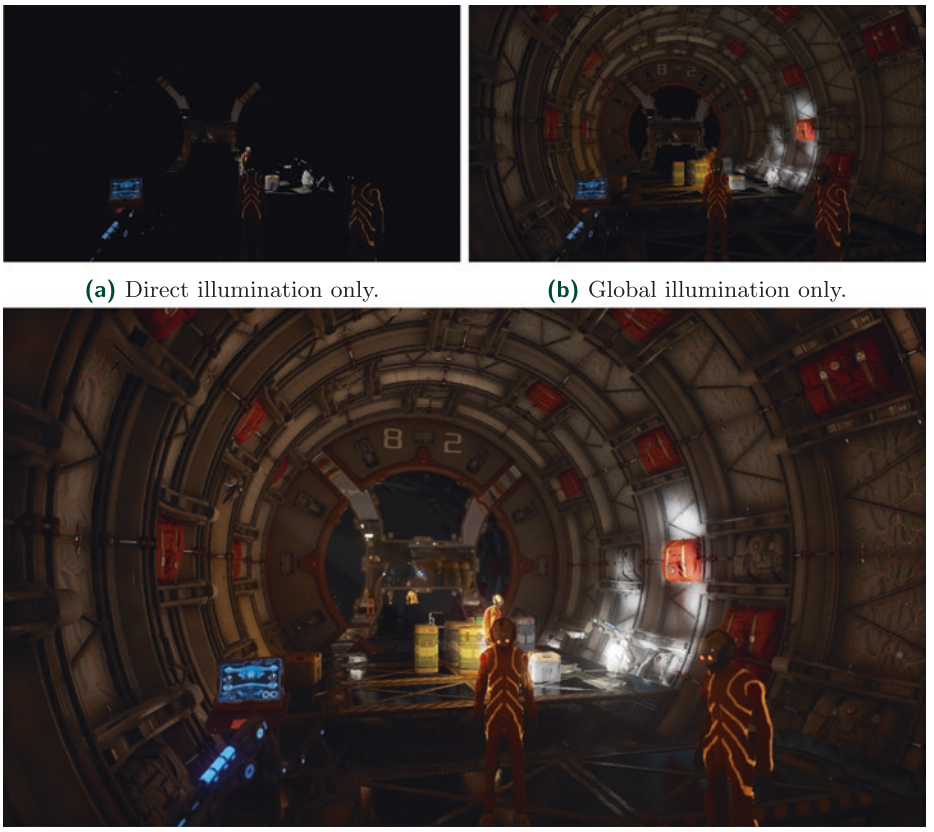
Figure 24-7. *Conference Room test scene.*



(a) Direct illumination only. **(b)** Global illumination only.

(c) Final result.

Figure 24-8. *Modified Sponza test scene.*



(a) Direct illumination only.

(b) Global illumination only.

(c) Final result.

Figure 24-9. *A section of the 3DMark Port Royal ray tracing test.*

Table 24-3 reports the computation times in milliseconds for these scenes with high-quality settings: 1080p resolution, three million initial photons, three bounces of indirect light, and four iterations of the spatial filter. The results were measured using an NVIDIA RTX 2080 Ti. Note that, for all scenes, the most costly phase is splatting. Time spent on filtering is roughly the same for all scenes, since it is independent of the scene's geometric complexity but is an image-space operation.

Table 24-3. *Performance of our photon mapping implementation for each scene on an NVIDIA RTX 2080 Ti, with times measured in milliseconds.*

Scene	RSM	Tracing	Splatting	Filtering	Total
Conference Room	1.6	5.2	7.5	3.3	**17.6**
Sponza	2.1	3.0	5.5	3.6	**14.2**
3DMark Port Royal	2.1	8.0	8.3	3.3	**21.7**

In Table 24-4 we examine the effect of varying some of the parameters. As would be expected, the time spent on RSMs, tracing rays, and photon splatting increases with the number of photons traced. Due to path termination from Russian roulette, increasing the number of bounces reduces performance less than adding a corresponding number of initial photons. Increasing image resolution correspondingly increases both splatting and filtering time.

Table 24-4. *Performance of the photon mapping algorithm in the Sponza scene with different settings, measured in milliseconds. Filtering is done with four spatial iterations. The baseline is set to what we would consider "low" settings for photon mapping: one million photons and a single bounce.*

Photons	Bounce	Res.	RSM	Tracing	Splatting	Filtering	Total
1 M	1	1080p	1.4	0.7	1.2	3.1	**6.1**
1 M	1	1440p	1.4	0.7	1.6	5.6	**9.3**
2 M	1	1080p	1.8	1.3	2.3	3.1	**9.0**
3 M	1	1080p	2.1	1.8	3.9	3.1	**10.8**
1 M	3	1080p	1.4	1.3	2.1	3.1	**7.9**

24.6 FUTURE WORK

There are a number of areas where performance or quality of the approach described here could be improved.

24.6.1 OPTIMIZING IRRADIANCE DISTRIBUTION BY SKIPPING SPLATTING

With high-density functions, the screen-space size of the splatting kernel can approach the size of a pixel, which makes drawing the splatting kernel wasteful. This could possibly be solved by writing the irradiance value directly to the framebuffer instead of splatting.

24.6.2 ADAPTIVE CONSTANTS FOR VARIANCE CLIPPING OF THE DETAIL COEFFICIENTS

Unfortunately, we cannot determine if the variance in the irradiance is caused by the low sample count or an actual difference in lighting conditions. This is partly mitigated by the larger sample set provided by stratified sampling. As these samples are accumulated using temporal filtering, the noise becomes visible in cases where temporal samples are being rejected. Therefore, it would be preferable to use less constricting variance clipping boundaries for these areas. Such a system could be implemented by scaling the variance clipping constant based on the weights that we use to define the accumulation of the temporal samples.

REFERENCES

[1] Clarberg, P., Jarosz, W., Akenine-Möller, T., and Jensen, H. W. Wavelet Importance Sampling: Efficiently Evaluating Products of Complex Functions. *ACM Transactions on Graphics 24*, 3 (2005), 1166–1175.

[2] Dachsbacher, C., and Stamminger, M. Reflective Shadow Maps. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games* (2005), pp. 203–231.

[3] Dachsbacher, C., and Stamminger, M. Splatting Indirect Illumination. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games* (2006), ACM, pp. 93–100.

[4] Dammertz, H., Sewtz, D., Hanika, J., and Lensch, H. Edge-Avoiding À-Trous Wavelet Transform for Fast Global Illumination Filtering. In *Proceedings of High-Performance Graphics* (2010), pp. 67–75.

[5] Heitz, E., and d'Eon, E. Importance Sampling Microfacet-Based BSDFs using the Distribution of Visible Normals. *Computer Graphics Forum 33*, 4 (2014), 103–112.

[6] Heitz, E., Hill, S., and McGuire, M. Combining Analytic Direct Illumination and Stochastic Shadows. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2018), pp. 2:1–2:11.

[7] Jensen, H. W. *Realistic Image Synthesis Using Photon Mapping*. A K Peters, 2001.

[8] Karis, B. High-Quality Temporal Supersampling. Advances in Real-Time Rendering in Games, SIGGRAPH Courses, 2014.

[9] Mara, M., Luebke, D., and McGuire, M. Toward Practical Real-Time Photon Mapping: Efficient GPU Density Estimation. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2013), pp. 71–78.

[10] McGuire, M., and Luebke, D. Hardware-Accelerated Global Illumination by Image Space Photon Mapping. In *Proceedings of High-Performance Graphics* (2009), pp. 77–89.

[11] O'Donnell, Y. Precomputed Global Illumination in Frostbite. Game Developers Conference, 2018.

[12] Salvi, M. An Excursion in Temporal Supersampling. From the Lab Bench: Real-Time Rendering Advances from NVIDIA Research, Game Developers Conference, 2016.

[13] Schied, C., Kaplanyan, A., Wyman, C., Patney, A., Chaitanya, C. R. A., Burgess, J., Liu, S., Dachsbacher, C., Lefohn, A., and Salvi, M. Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination. In *Proceedings of High-Performance Graphics* (2017), pp. 2:1–2:12.

[14] Schregle, R. Bias Compensation for Photon Maps. *Computer Graphics Forum 22*, 4 (2003), 729–742.