CHAPTER 32

Accurate Real-Time Specular Reflections with Radiance Caching

Antti Hirvonen, Atte Seppälä, Maksim Aizenshtein, and Niklas Smal

UL Benchmarks

ABSTRACT

We present an algorithm for perspective-correct, real-time specular illumination for surfaces of varying glossiness in dynamic environments. Our algorithm leverages properties from earlier techniques (e.g., radiance probes and screen-space reflections) while reducing the amount of visual errors by adding ray tracing to the rendering pipeline. Our algorithm extends previous work by allowing accurate reflections for all surfaces regardless of the material, and it has global coherence (i.e., there are no visible discontinuities). With radiance caching, multiple samples can be efficiently computed as the radiance computation is decoupled from the final shading. The radiance cache is also used to approximate the specular term for the roughest surfaces without any ray tracing.

32.1 INTRODUCTION

Real-time rendering engines approximate lighting computations due to the computational cost of accurate simulations. Lighting can be quickly evaluated only for idealized or nearly idealized light sources such as point lights. However, illumination is a global phenomena and it can be affected significantly by light reflected from surfaces and light emitted from complex sources. Simulation of these components is usually expensive, but both have to be taken into account for realistic lighting. The aggregated contribution of such terms is known as *global illumination*. In real-time graphics, most terms of global illumination are usually precomputed.

Rendering engines commonly split the surface into two separate layers that contribute to the illumination: diffuse and specular. Each layer is composed of microscopic, flat area elements called *microfacets* which are described by a distribution rather than geometrical modeling. The mean slope (or in some cases the standard deviation of the slope) is described by a surface parameter called *roughness*.

Diffuse layers describe weak correlation between the scattering distribution and the incoming light direction. A diffuse microfacet scatters luminous energy proportionally to the cosine of the angle between the incoming light direction and the microfacet normal direction. A diffuse material that exhibits flat micro-structure is called *Lambertian*. In the case of diffuse illumination, the response mostly depends on the total irradiance on the surface. Therefore, the actual distribution of the incoming light does not have to be known in order to compute the radiance scattered in some direction. This observation is the key idea behind precomputed *irradiance caches* such as light maps or irradiance probes. Due to the low-frequency nature of the input data, the irradiance component can be packed aggressively and stored efficiently to cover the entire scene. Furthermore, minor changes in the scene's direct diffuse illumination do not significantly affect the indirect diffuse term.

The second term, specular illumination, describes strong correlation between the scattering distribution and the light's incoming direction. Every specular microfacet reflects light according to Snell's law, and the reflected energy of the light is determined by Fresnel equations. A surface that exhibits flat micro-structure with specular microfacets is an *idealized mirror*. However, materials are rarely perfect mirrors and they scatter light into some preferred set of directions instead of just one: such surfaces are classified as *glossy specular*. By Helmholtz reciprocity, the measured radiance depends on a set of incoming radiances, taking a wider distribution into account when materials are rougher. The specular term is also referred to as *reflection* later in this chapter.

These observations make it impractical to store many radiance samples regardless of the data structure. Therefore, current rendering engines usually just capture radiance from a few points in the scene, or use already computed main camera radiance for the specular environment term. These approximations have their own shortcomings, which are analyzed briefly in Section 32.2. The only practical way to compute an accurate specular term during runtime is to actually sample radiance from the scene for each shaded point.

In this chapter, we present an algorithm for efficient computation of the indirect specular term for surfaces of varying glossiness regardless of the scene. We use the new Microsoft DirectX Raytracing (DXR) pipeline, as introduced into DirectX 12, to query global surface visibility in the scene for a set of rays defined by the specular BRDF. Radiance for these rays can be efficiently computed with our cached approach. Our algorithm also enables efficient specular term approximation for rough surfaces for which the view-dependent variance is low. The end result after post-filtering provides accurate and real-time specular illumination estimates for each pixel on the screen. See Figure 32-1.

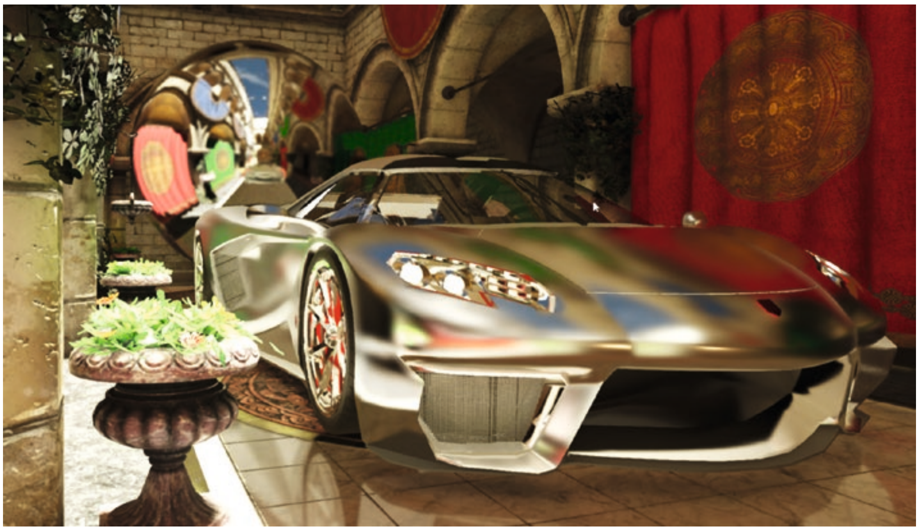


Figure 32-1. *A glossy car body, reflective floor, and mirror ball in the rear pick up local reflections at interactive rates.*

32.2 PREVIOUS WORK

Traditional and widely used techniques for reflections include planar reflections, screen-space reflections, and various image-based lighting approaches.

32.2.1 PLANAR REFLECTIONS

Planar reflections are simple to produce but require rendering the scene geometry multiple times—once for each planar reflector. Depending on the scene and the engine in question, this can be a costly operation on CPU, GPU, or both. Planar reflections only work well for planar or near-planar reflectors. Reflections of rough surfaces are problematic because planar reflectors cannot capture radiance except in the direction of the virtual camera.

32.2.2 SCREEN-SPACE REFLECTIONS

Screen-space reflections (SSR) is a reflection technique that only uses screen-space data to approximate the specular term for the visible surfaces. The main idea is to cast one or more rays in screen space according to the specular BRDF of the surface and approximate radiance for those rays from the main camera illumination buffers. For each ray, a hit position is computed from the depth buffer data using ray marching. This makes SSR an incredibly cheap technique because no complex input data are required, which makes it viable even on lower-end hardware. Dynamic scenes are naturally supported without any extra cost. See the work of McGuire and Mara [12] and Stachowiak [16] for more information.

Unfortunately, SSR has multiple downsides. First, as it only operates on the screen-space data, occlusion can be incorrectly interpreted based on the depth buffer. In such cases, a ray can either terminate too early or pass through objects that it should actually hit. Second, main camera radiance naturally has only a single layer, and thus objects occluded in the view of main camera or outside of the camera frustum are not seen in reflections.

32.2.3 IMAGE-BASED LIGHTING

Image-based lighting (IBL) techniques approximate illumination from some captured imagery, stored usually in radiance probes that encode a spherical radiance map (also known as radiance cubes, reflection cubes, or reflection probes). Each probe can be associated with a proxy geometry object, such as a sphere or a box, that gives an approximated hit point in the scene [11]. Probes are also usually prefiltered to allow fast approximation of glossy materials and can be either precomputed or updated in real time depending on the frame budget. Readers can refer to Debevec's work [3] for more information on IBL in general.

32.2.4 HYBRID APPROACHES

Multiple reflection techniques are usually combined to produce the final image. For example, screen-space reflections can be used in conjunction with the offline-generated radiance probes to create an approximate real-time specular illumination [5]. However, mixing various techniques can lead to visible discontinuities in the final illumination at the places where the reflection technique switches.

32.2.5 MISCELLANEOUS

More recent approaches have higher quality, but they come with an added computational cost. Voxel cone tracing can produce realistic specular terms even in dynamic scenes as presented by Crassin et al. [1], but it operates on the voxel scale. The approach presented by McGuire et al. [13] allows computation of accurate indirect diffuse and specular illumination from a set of precomputed light probes. These probes are augmented with a depth buffer for computing the intersection with a similar ray marching routine as in screen-space reflections. However, the technique is not fully dynamic. Neither of these techniques are yet widely used in rendering engines.

32.3 ALGORITHM

Based on the previous work and general observations of modern rendering engines, the design of our algorithm stems from the following observations:

> Screen-space reflections effectively approximate the local specular term and produce realistic results when there are no discontinuities in the final illumination, i.e., when the neighboring texels successfully sample from the screen space. However, discontinuities can immediately appear when the radiance is computed by other means (such as by sampling from a radiance cube). The rest of the reflection pipeline must match with the screen-space data to remove these discontinuities. Reusing the screen-space data also reduces the amount of costly radiance recomputations.

> Only smooth, mirror-like surfaces need a high-resolution render. Lower-resolution approximations are fine for reflections of rougher surfaces as results are averaged over a set of directions.

> Rays that are traced over a set of surfaces may hit approximately the same points in the scene. This becomes more likely as the number of radiance samples per pixel is increased.

> It is common for game environments to have a small number of dynamic objects.

In practice, our algorithm enhances previous screen-space reflection and radiance probe techniques with ray tracing. Our novel contributions include the way we combine these techniques, the heuristics we define for sampling the probes, and modifications to motion vectors for temporal reflection filtering.

Figure 32-2 shows the various stages of our algorithm integrated into a traditional deferred rendering pipeline. The green parts show the steps of a simple traditional deferred rendering pipeline, and the purple parts are the additions for our implementation of ray traced reflections. The added parts comprise creation of the radiance cache for static geometry, lighting of the radiance cache, radiance sample generation, and reflection filtering. Our radiance cache is created as a preprocessing step for the static geometry. Lighting of the radiance cache can be seen as decoupled shading for the reflection ray tracing and sampling passes, and rays not found from the cache are simply shaded using material and light information as in a normal ray tracer. After a radiance value has been computed for all rays traced from the visible texels, a spatiotemporal filter is applied, and the filtered result is combined with the diffuse and direct specular surface illumination. Effects such as volumetric lighting are only applied to the final illumination after the reflections have been fully resolved. This is necessary to reduce illumination discontinuities as the sampled screen-space illumination must match with the radiance cache and fully shaded rays. The world-space clustering pass plays an important role; as rays can hit any point in the scene, a world-space data structure can be used to accelerate lighting without evaluating all lights in a scene.

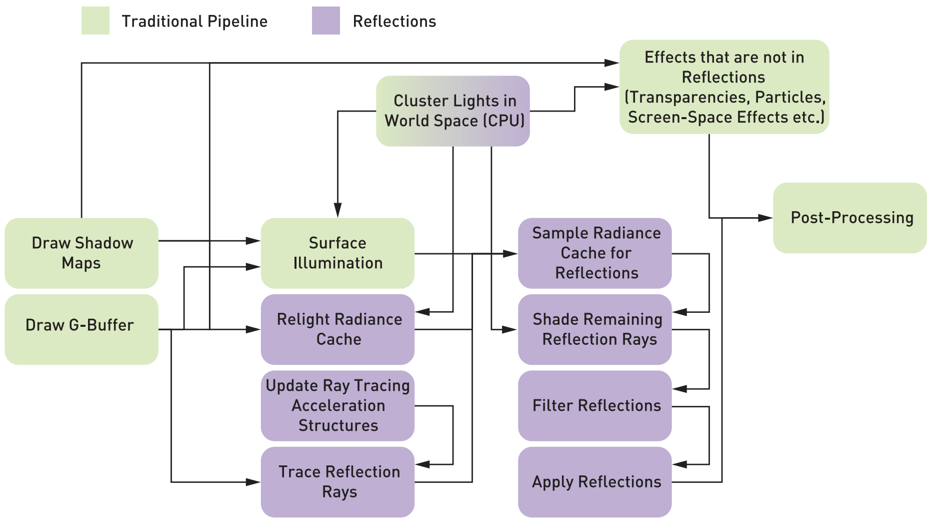

Traditional Pipeline
Reflections
Effects that are not in Reflections (Transparencies, Particles, Screen-Space Effects etc.)
Cluster Lights in World Space (CPU)
Post-Processing
Draw Shadow Maps
Surface Illumination
Sample Radiance Cache for Reflections
Draw G-Buffer
Relight Radiance Cache
Shade Remaining Reflection Rays
Update Ray Tracing Acceleration Structures
Filter Reflections
Trace Reflection Rays
Apply Reflections

Figure 32-2. *Stages and data flows of the overall rendering pipeline and their dependencies.*

Figure 32-3 shows how the screen-space illumination texture and radiance probes can be used to sample radiance from intersection points computed by the ray tracing pipeline using our technique. The intersection of ray R_2 is visible on screen. Radiance probe 1 sees the intersections of rays R_1, R_2, and R_3, and radiance probe 2 sees the intersection of ray R_1. For all these rays the radiance can be sampled from caches. In contrast, the intersection of ray R_4 is unavailable in the two probes or screen-space data, and therefore it has to be explicitly shaded. While the radiance probes themselves must be shaded, multiple rays may use the same precomputed value, which gives a great benefit when the shading is complex and there are glossy surfaces that do not require a large resolution for the sampled radiance probes. Furthermore, the shading of the radiance probes has the benefit of locality; neighboring pixels are likely to compute the same lights, and the materials are coherently sampled from the probe's G-buffer. These factors make the cache illumination efficient to compute on a modern GPU.

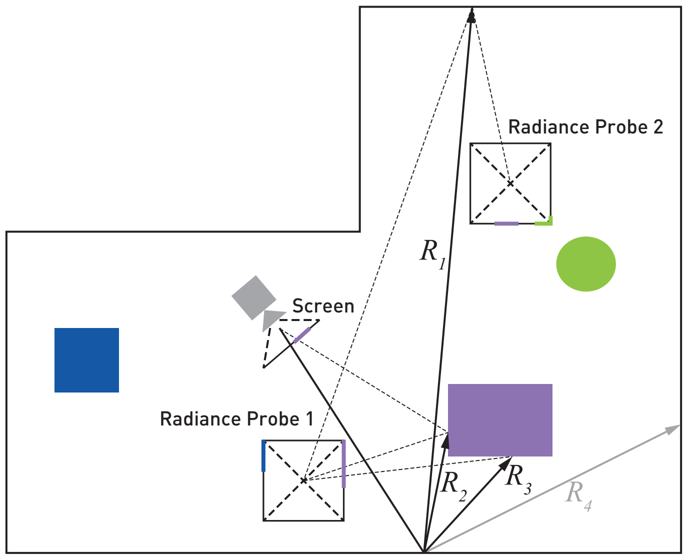


Figure 32-3. *Visualization of the cache sampling strategy for multiple reflection rays from a glossy reflective surface.*

32.3.1 RADIANCE CACHE

Our cache entries, i.e., the radiance probes, are stored as cube maps. As cube maps are native entities in the common graphics APIs, they are easy to render and sample. We aim to study other mappings, such as octahedral projection used by McGuire et al. [13], as future work. Contrary to previous approaches, we do not prefilter the probes at all. All filtering runs in screen space.

Similarly to earlier techniques, radiance probes must be placed in the scene either automatically or manually in a way that they cover most of the scene surfaces. In our case, probes were placed manually by artists to locations where visibility is maximized and overlaps are minimized. Automatic placement is another avenue for future work.

32.3.1.1 RENDERING

We render only static geometry into our radiance cache. This allows us to separate the rasterization of the geometry into a precomputed pass, thus removing all runtime geometry processing load from both CPU and GPU. Runtime GPU workload is reduced to a deferred illumination pass. Each radiance probe in our system is composed of a full G-buffer texture set: albedo, normal, roughness, metalness, and luminance. All these textures are required for lighting the cache samples.

32.3.1.2 LIGHTING

The lighting pass evaluates all direct lights for the cache samples. We use the same compute pass for radiance cache illumination as for the main camera illumination. Each full probe in the current system is reilluminated each frame. This can be optimized further by illuminating only those areas in the cache that were hit by rays; see Section 32.7 for more information. Our world-space light clustering algorithm effectively culls lights for the compute pass regardless of probe position. We use the same light clustering scheme for the main camera illumination as well.

One important thing to note about cache lighting is that the view during lighting is fixed to that of the main camera. Albeit wrong, this makes the lighting match with the main camera illumination, thus removing any seams that might arise when combining the screen-space hits with the cache hits or fully shaded rays. The view mostly affects the specular term of direct lighting.

32.3.2 RAY TRACING

The main ray tracing pass in our algorithm is responsible for generating the sample directions according to our specular BRDF, tracing the rays, and storing the hit information for the later passes that actually compute the radiance for the set of rays.

32.3.2.1 SAMPLING THE SPECULAR BRDF

The incoming light caused by specular reflection toward the view direction ω_o at point X with geometric normal ω_g is given by the rendering equation:

$$L_o(X, \omega_o) = \int_{\Omega_i} L_i(X, \omega_i) f_s(\omega_i, \omega_o)(\omega_i \cdot \omega_g) d\omega_i, \tag{1}$$

where Ω_i is the positive hemisphere on the point X, ω_i are the directions taken from that hemisphere, and for the BRDF f_s, we use the Cook-Torrance model with GGX distribution of microfacet normals. This may be computed using Monte Carlo integration with importance sampling as

$$L_o(X, \omega_o) \approx \frac{1}{n} \sum_{i=1}^{n} \frac{L_i(X, \omega_i) f_s(\omega_i, \omega_o)(\omega_i \cdot \omega_g)}{f_{\Omega_i|\Omega_o}(\omega_i)}, \tag{2}$$

where $f_{\Omega_i|\Omega_o}$ is the sampling probability density function. To sample f_s, we utilize the GGX distribution of visible normals using the exact sampling routine introduced by Heitz [7] and precomputed Halton sequences [6] of bases 2 and 3 as input for the sampling routine. However, instead of directly using the approximation in Equation 2, we follow the same variance reduction scheme as proposed by Stachowiak [16, 17] by dividing and multiplying by the same factor $\int_{\Omega_i} f_s(\omega_i, \omega_o)(\omega_i \cdot \omega_g) d\omega_i$ and discretizing the denominator:

$$L_o(X, \omega_o) \approx \frac{\sum_{i=1}^{n} \frac{L_i(X, \omega_i) f_s(\omega_i, \omega_o)(\omega_i \cdot \omega_g)}{f_{\Omega_i|\Omega_o}(\omega_i)}}{\sum_{i=1}^{n} \frac{f_s(\omega_i, \omega_o)(\omega_i \cdot \omega_g)}{f_{\Omega_i|\Omega_o}(\omega_i)}} \int_{\Omega_i} f_s(\omega_i, \omega_o)(\omega_i \cdot \omega_g) d\omega_i. \tag{3}$$

The term $\int_{\Omega_i} f_s(\omega_i, \omega_o)(\omega_i \cdot \omega_g) d\omega_i$ is a function of $\omega_o \cdot \omega_g$, the roughness, and the reflectance at the incident angle (base reflectance). When Schlick's approximation [15] is used instead of the full Fresnel term, the base reflectance can be factored out of the integral, and the BRDF integral over the hemisphere can be approximated by a rational function. We derived such an approximation using numerical error minimization in Mathematica and arrived at

$$\begin{aligned}
&\int_{\Omega_i} f_s(\omega_i, \omega_o)(\omega_i \cdot \omega_g) d\omega_i \\
&\approx \frac{\begin{pmatrix} 1 & \alpha \end{pmatrix} \begin{pmatrix} 0.99044 & -1.28514 \\ 1.29678 & -0.755907 \end{pmatrix} \begin{pmatrix} 1 \\ (\omega_o \cdot \omega_g) \end{pmatrix}}{\begin{pmatrix} 1 & \alpha & \alpha^3 \end{pmatrix} \begin{pmatrix} 1 & 2.92338 & 59.4188 \\ 20.3225 & -27.0302 & 222.592 \\ 121.563 & 626.13 & 316.627 \end{pmatrix} \begin{pmatrix} 1 \\ (\omega_o \cdot \omega_g) \\ (\omega_o \cdot \omega_g)^3 \end{pmatrix}} \\
&+ \frac{\begin{pmatrix} 1 & \alpha \end{pmatrix} \begin{pmatrix} 0.0365463 & 3.32707 \\ 9.0632 & -9.04756 \end{pmatrix} \begin{pmatrix} 1 \\ (\omega_o \cdot \omega_g) \end{pmatrix}}{\begin{pmatrix} 1 & \alpha & \alpha^3 \end{pmatrix} \begin{pmatrix} 1 & 3.59685 & -1.36772 \\ 9.04401 & -16.3174 & 9.22949 \\ 5.56589 & 19.7886 & -20.2123 \end{pmatrix} \begin{pmatrix} 1 \\ (\omega_o \cdot \omega_g)^2 \\ (\omega_o \cdot \omega_g)^3 \end{pmatrix}} R_0,
\end{aligned} \tag{4}$$

where α is the square of the linear roughness in the GGX model and R_0 is the reflectance from a direction parallel to the normal. This technique has the further advantage of preserving details related to some surface properties since the pre-integrated term is noise free and does not need to be filtered at all. Equation 4 provides a fast way of evaluating this integral. Another way is to tabulate the function and perform a lookup in that table [10].

32.3.2.2 RAY GENERATION AND HIT STORAGE

In our algorithm the actual ray tracing part is simple because the computation of radiance is separated from the tracing of rays. The ray tracing pipeline is only used to find the correct scene intersection point. Pseudocode for both ray generation and hit shaders are given in Listing 32-1.

A ray is generated using the importance sampled direction, and the surface position reconstructed from G-buffer depth is used as the origin. Rays are not generated for materials with a roughness value of over `RT_ROUGHNESS_THRESHOLD`; for such materials the radiance is sampled from the cache with just a direction vector. For traced rays the ray length, barycentric coordinates, instance index, and primitive index of the resulting hit are written to a texture, but no further work is required in this pass. Geometry data is stored because the term $L_i(x, \omega_i)$ is not always found in the screen-space radiance or the radiance cache, and it has to be computed using the correct material. Note that our implementation supports a single material per instance, hence the instance index uniquely identifies the used material. Implementations with multiple materials per instance will need to write out more data.

Listing 32-1. *Ray generation and hit shaders.*

```
void rayHit(inout Result result)
{
  result.RayLength = RayTCurrent();
  result.InstanceId = InstanceId();
  result.PrimitiveId = PrimitiveIndex();
  result.Barycentrics = barycentrics;
}

void rayGen()
{
  float roughness = LoadRoughness(GBufferRoughness);
  uint sampleCount = SamplesRequired(roughness);
```

```
 if (roughness < RT_ROUGHNESS_THRESHOLD) {
  float3 ray_o = ConstructRayOrigin(GBufferDepth);
  for (uint sampleIndex = 0;
     sampleIndex < sampleCount; sampleIndex++) {
   float3 ray_d = ImportanceSampleGGX(roughness);

   TraceRay(ray_o, ray_d, results);
   StoreRayIntersectionAttributes(results, index.xy, sampleIndex);
   RayLengthTarget[uint3(index.xy, sampleIndex)] = rayLength;
  }
 }
}
```

32.3.3 RADIANCE COMPUTATION FOR RAYS

As mentioned in Section 32.3.2.1, we use a variance reduction scheme in which the stochastic sampling result is divided by the sum of the weights of each radiance sample and the result is later multiplied by the approximation of the BRDF integral over the hemisphere. Applying a shorthand notation to Equation 3, so that L_{total} is the sum of the weighted radiance samples and w_{total} is the sum of the sample weights for a single pixel, the total radiance from specular reflection can be written as

$$L_o\left(X, \omega_0\right) \approx \frac{L_{\text{total}}}{w_{\text{total}}} \int_{\Omega_i} f_s\left(\omega_i, \omega_o\right)\left(\omega_i \cdot \omega_g\right) d\omega_i. \tag{5}$$

Similar to Stachowiak's work [16], all of the terms are combined only after spatiotemporal filtering because denoising a ratio estimator directly would not make the approximation converge toward the correct result [9]. Therefore, the per-pixel sums L_{total} and w_{total} are written to separate textures by the radiance cache sampling pass and the ray shading pass: first, the cache sampling pass writes the terms for all rays that were present in the cache, then the ray shading pass accumulates L_{total} and w_{total} for rays that did not have a radiance sample in the cache. The implementation of the cache sampling and ray shading passes is discussed in the subsequent sections.

32.3.3.1 SAMPLING THE RADIANCE CACHE AND SCREEN-SPACE ILLUMINATION

Since the radiance computed into the cache and screen-space illumination match, they can both be used to approximate the radiance $L_i(x, \omega_i)$. The importance sampled direction ω_i can be regenerated to obtain the same direction as in the ray tracing pass, and the intersection point can be computed from the direction vector, G-buffer, and ray length written by the ray tracing pass. To sample the main camera illumination texture, the intersection point is projected to screen space and sampling continues with the obtained texel coordinates. The validity of the radiance sample is checked by comparing the screen-space G-buffer depth against the computed depth. If sampling fails, then any of the radiance probes can be sampled using a world-space direction vector from the cube map toward this ray intersection, but certain thresholds, described later in this section, are needed to ensure the correctness of the sample.

An outline of the sampling pass is shown in Listing 32-2. Note that for materials with roughness above a certain threshold (RT_ROUGHNESS_THRESHOLD), we use a proxy geometry intersection to generate the hit position and sample the radiance probes using that direction.

Listing 32-2. *Routine for sampling precomputed radiance.*

```
void SamplePrecomputedRadiance()
{
  float roughness = LoadRoughness(GBufferRoughness);
  float3 rayOrigin = ConstructRayOrigin(GBufferDepth);
  float3 L_total = float3(0, 0, 0); // Stochastic reflection
  float3 w_total = float3(0, 0, 0); // Sum of weights
  float primaryRayLengthApprox;
  float minNdotH = 2.0;
  uint cacheMissMask = 0;

  for (uint sampleId = 0;
        sampleId < RequiredSampleCount(roughness); sampleId++) {
    float3 sampleWeight;
    float NdotH;
    float3 rayDir =
          ImportanceSampleGGX(roughness, sampleWeight, NdotH);
    w_total += sampleWeight;
    float rayLength = RayLengthTexture[uint3(threadId, sampleId)];
    if (NdotH < minNdotH)
    {
      minNdotH = NdotH;
      primaryRayLengthApprox = rayLength;
    }
    float3 radiance = 0; // For cache misses, this will remain 0.
    if (rayLength < 0)
      radiance = SampleSkybox(rayDir);
    else if (roughness < RT_ROUGHNESS_THRESHOLD) {
      float3 hitPos = rayOrigin + rayLength * rayDir;
      if (!SampleScreen(hitPos, radiance)) {
        uint c;
        for (c = 0; c < CubeMapCount; c++)
          if (SampleRadianceProbe(c, hitPos, radiance)) break;
        if (c == CubeMapCount)
          cacheMissMask |= (1 << sampleId); // Sample was not found.
      }
    }
    else
      radiance = SampleCubeMapsWithProxyIntersection(rayDir);
    L_total += sampleWeight * radiance;
  }

  // Generate work separately for misses
  // to avoid branching in ray shading.
  uint missCount = bitcount(cacheMissMask);
  AppendToRayShadeInput(missCount, threadId, cacheMissMask);
  L_totalTexture[threadId] = L_total;
  w_totalTexture[threadId] = w_total;
  // Use ray length of the most likely ray to approximate the
  // primary ray intersection for motion.
  ReflectionMotionTexture[threadId] =
        CalculateMotion(primaryReflectionDir, primaryRayLengthApprox);
}
```

Pseudocode for sampling a single probe is given in Listing 32-3.

Listing 32-3. *Routine for sampling a single probe.*

```
bool SampleRadianceProbe(uint probeIndex,
                         float3 hitPos,
                         out float3 radiance)
{
  CubeMap cube = LoadCube(probeIndex);
  float3 fromCube = hitPos - cube.Position;
  float distSqr = dot(fromCube, fromCube);
  if (distSqr <= cube.RadiusSqr) {
    float3 cubeFace = MaxDir(fromCube); // (1,0,0), (0,1,0) or (0,0,1)
    float hitZInCube = dot(cubeFace, fromCube);
    float p = ProbabilityToSampleSameTexel(cube, hitZInCube, hitPos);
    if (p < ResolutionThreshold) {
      float distanceFromCube = sqrt(distSqr);
      float3 sampleDir = fromCube / distanceFromCube;
      float zSeenByCube =
            ZInCube(cube.Depth.SampleLevel (Sampler, sampleDir, 0));
      // 1/cos(view angle), used to get the distance along the view ray
      float cosCubeViewAngleRcp = distanceFromCube / hitZInCube;
      float dist = abs(hitZInCube - zSeenByCube) * cosCubeViewAngleRcp;
      if (dist <
            OcclusionThresholdFactor * hitZInCube / cube.Resolution) {
        radiance = cube.Radiance.SampleLevel(Sampler, sampleDir, 0);
        return true;
      }
    }
  }
  return false;
}
```

The radius check is done to accelerate the computation, and it should be adjusted so that samples outside this radius are unlikely to exist or have enough detail. As a further optimization, clustering could be used to avoid the radius check the same way that it is used for point lights. The occlusion check is done to ensure that the sampled position corresponds to the actual hit position, since there could be an occluding geometry in front of the radiance probe, or the intersection could be in a dynamic object that is not present in the radiance probes. Since we have a separate check for the resolution, we define the distance threshold to allow variations in depth likely caused by the low resolution. We use the function $\beta \frac{z_c}{x_c}$, where z_c is the depth of the intersection in the cube, x_c is the cube resolution, and β is a constant that should be adjusted to be large enough to allow sampling from surfaces that are not perpendicular to the view ray of the reflection cube. Figure 32-4 shows an example of a reflection ray intersection that is not found from a cube map due to occlusion by another geometry.

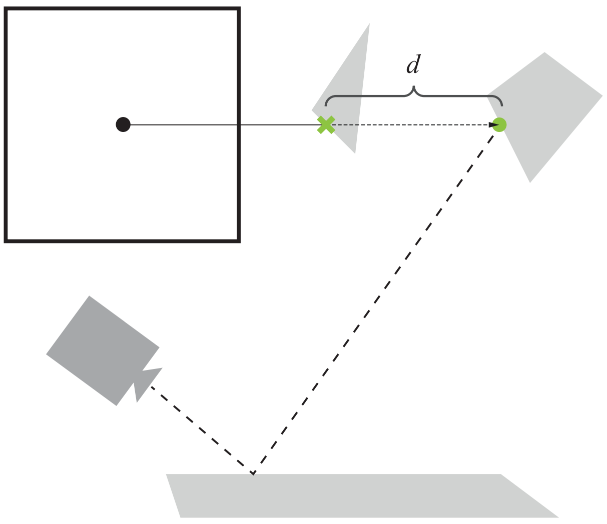


Figure 32-4. *Thresholds used for radiance probe sampling: distance between the actual intersection and the position found from a reflection cube.*

For defining the threshold for radiance probe resolution, we use a heuristic on how much error the finite resolution of the radiance probe may cause for the reflection direction, taking into account the distribution from which the direction is importance-sampled. To quantize this error, we analyze the probability of sampling points that are aliased to the same texel in the radiance probe (function `ProbabilityToSampleSameTexel` in Listing 32-3). Figure 32-5 shows a situation in which two of three rays from the same surface are aliased to a single sample in a radiance probe.

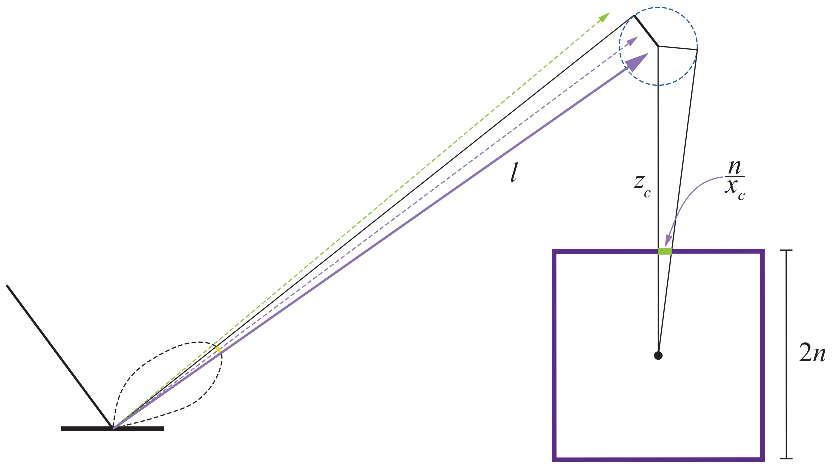


Figure 32-5. *Thresholds used for radiance probe sampling: visualization of the resolution threshold heuristic. The value* $\frac{n}{x_c}$ *is half a pixel in width in world space at the cube map's near plane at distance n. This value is then proportional to the radius of the circle sampled at* z_c.

The probability may be obtained by integrating the microfacet distribution function

$$f_{\Omega_m|\Omega_o} = \frac{G_1(\omega_o,\omega_m)D_{GGX}(\omega_m)(\omega_m\cdot\omega_o)}{(\omega_g\cdot\omega_o)} \tag{6}$$

over a region on the hemisphere that is centered at the exact microfacet normal and bounded by a region with a size that is derived from the spacing between pixels in the radiance probe. The sampled point (center of the circle in Figure 32-5) can be bound by a sphere that covers a single texel in the radiance probe. Assuming that the center of the sphere is located on the axis of the cube, then its radius is given by

$$r_{\text{ref}} = \frac{z_c}{\sqrt{1+x_c^2}} \approx \frac{z_c}{x_c}, \tag{7}$$

where z_c is the linear depth of the sample point and x_c is the resolution of the radiance probe's cube map. For cubes, the distance to the near plane, n, cancels out, thus not affecting the calculation. It is possible to generalize the calculation for off-axis sample points, but we are going to neglect it because in cube maps the error is only up to a finite constant from the approximation.

Now we need to evaluate the probability of a reflected ray hitting that sphere. While accurate approximations exist for integrating BRDFs over areas such as the sphere [8], in our case we need only a crude approximation that is efficient to compute. The reflection direction density over the reflected solid angle is given by

$$f_{\Omega_i|\Omega_o} = f_{\Omega_m|\Omega_o}\left|\frac{d\omega_m}{d\omega_i}\right| = \frac{G_1(\omega_o, \omega_m)D_{GGX}(\omega_m)}{4(\omega_g \cdot \omega_o)}. \tag{8}$$

With the assumption that the projected sphere's solid angle is small, we can approximate the probability of sampling the texel in the cube map:

$$\begin{aligned}\Pr(\omega_i \in S) &= \int_{S\subseteq\Omega_i} d\omega_i \frac{G_1(\omega_o, \omega_m)D_{GGX}(\omega_m)}{4(\omega_g \cdot \omega_o)} \\ &\approx \frac{G_1(\omega_o, \omega_m)D_{GGX}(\omega_m)}{(\omega_g \cdot \omega_o)}\frac{\pi}{2}\left(1-\sqrt{1-\left(\frac{z_c}{lx_c}\right)^2}\right) \\ &\approx \frac{G_1(\omega_o,\omega_m)D_{GGX}(\omega_m)}{(\omega_g \cdot \omega_o)}\frac{\pi}{4}\left(\frac{z_c}{lx_c}\right)^2,\end{aligned} \tag{9}$$

where l is the length of the reflected ray as in Figure 32-5. The first approximation is obtained by doing a single-sample Monte Carlo integration (some value in the domain multiplied by the integration volume, which is the solid angle subtended by a sphere), and the second approximation is obtained by taking the Taylor expansion of the square root term. The threshold can then be defined to anything between 0 and 1. For example, a threshold of 0.1 would mean that if the probability of sampling the texel is 10% or higher, then the cube is rejected, because a single texel does not contain the high-frequency information needed to reconstruct the reflection. However, if the probability is low enough, then the texel is sufficient to reconstruct reflection information. The latter is generally the case for highly rough surfaces, or when the sampled direction is on the tail end of the D_{GGX} distribution. Note that for perfect and near-perfect mirrors this threshold is almost never satisfied, but due to the finite resolution of the view, the samples may still be acceptable. Therefore, when computing the threshold, we clamp the surface's roughness α to an adjustable minimum value to allow sampling from cubes that have a relatively high resolution. As future work the curvature of the surface and view resolution itself could also be taken directly into account.

32.3.3.2 SHADING CACHE-MISSED RAYS

Covering the whole scene with radiance probes so that every point is visible in some probe would require an extremely high amount of probes in a practical scene, and each one adds overhead to the sampling and relighting passes. Further, we do not include dynamic geometry in the probes, and the resolution of the probes may be too low for some rays, especially for highly smooth surfaces. Therefore, we still need a robust way to reconstruct the radiance for those ray intersections that are not visible in any probe nor in the screen-space illumination texture.

As a fallback we compute the radiance for each of the unshaded samples using a separate compute pass. As the ray tracing pass writes out the geometry instance index, primitive index, and barycentric coordinates, these can be directly used to construct the accurate hit point and query all required data for the illumination pass. Although now it would be possible to use the accurate ray direction for specular highlights, we use the camera direction here to match the specular illumination computed for the radiance cache and screen-space illumination.

To avoid branching within warps/wavefronts based on how many samples require shading, we compact the indices of rays that were cache misses into a separate buffer in the sampling pass. Another compute pass then applies the radiance computation for each of the required rays read from the buffer, so that each thread within a warp/wavefront has the same amount of work to execute. This is essential because for glossy reflections the directions between pixels have high variance, so the cache misses are scattered randomly within large areas and some warps/wavefronts would execute the radiance computation only because one or a few lanes had cache misses.

32.4 SPATIOTEMPORAL FILTERING

The described algorithm provides a crude approximation of the specular environment illumination term. However, due to the low sample counts—one sample per pixel in the extreme case—the resulting approximation is noisy. Therefore, the resulting radiance estimates must be aggressively filtered both spatially and temporally to get rid of the high-frequency noise that results from undersampling the rendering integral. The amount of observed noise depends on the surface attributes and the distribution of light in the scene.

In this section, we describe a filtering scheme that we used to generate the results seen in Section 32.5. As filtering is not the main topic of this chapter, only a short description with references is provided. In practice the noise from the reflection pass is similar to that in path tracers, and any algorithm suited for cleaning up path traced images will work here as well. Note that, similar to Heitz et al. [9], the filtering process is applied separately for both terms of the ratio estimator that we use, and the combination of the terms is done only after filtering, as mentioned in Section 32.3.3.1.

32.4.1 SPATIAL FILTERING

With spatial filtering, we aim to compensate for the low sample count by sharing samples over the pixel neighborhood. Samples are shared only if the neighboring pixels match in surface attributes. Our spatial filter is based on the edge-avoiding Á-Trous wavelet transform [2] that we enhanced with specific reflection-related weight functions. We perform multiple iterations of the Á-Trous wavelet transformation, where each iteration generates a set of scaling coefficients. These coefficients provide a low-pass representation of the kernel footprint without the undesired high-frequency noise. The transformation uses previous coefficients as an input for the following iteration step. This allows us to accumulate filtered samples efficiently over a large screen-space area while the weight functions suppress invalid samples. See Figure 32-6.

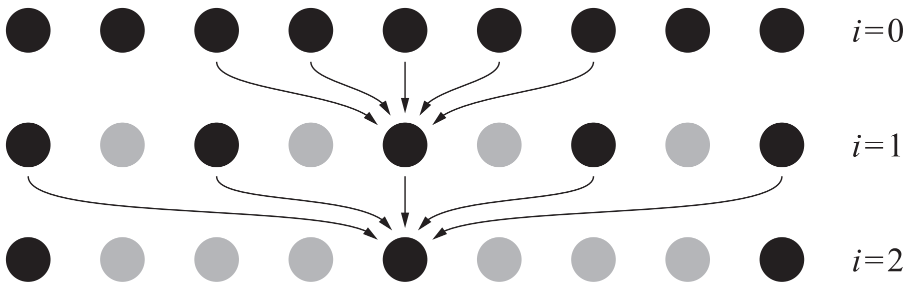


Figure 32-6. *Three iterations of one-dimensional stationary wavelet transform while the kernel footprint increases exponentially. Arrows show the nonzero pixels of previous result contributing to the current result, while the gray dots are pixels with zero value. Figure after Dammertz et al. [2].*

Our implementation follows the previous work by Dammertz et al. [2] and Schied et al. [14]. Each wavelet iteration is performed as a 5 × 5 cross-bilateral filter. The contributing samples are weighted by a function $w(P, Q)$, where P is the current pixel and Q the contributing sample pixel from the sample neighborhood. We calculate the scaling coefficient S_{i+1} as

$$S_{i+1} = \frac{\sum_{Q\in\Omega} h(Q)\,w(P,Q)\,S_i(Q)}{\sum_{Q\in\Omega} h(Q)\,w(P,Q)}, \tag{10}$$

where $h(Q) = \left(\frac{1}{8}, \frac{1}{4}, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}\right)$ is the filter kernel. The weight function $w(P, Q)$ controls the contribution of the sample Q based on the G-buffer attributes of that sample. The components contributing to this weight function can be categorized into four groups: edge-stopping, roughness, reflection-direction, and ray-length. These four groups are discussed in following sections.

To simplify the weight functions, we define a function f_w as a smooth interpolation function between limits a and b as

$$f_w(a, b, x) = 1 - \text{smoothstep}(a, b, x), \tag{11}$$

where `smoothstep` is the standard cubic Hermite interpolator as provided by shading languages.

32.4.1.1 EDGE-STOPPING WEIGHT

Edge-stopping weights prevent the distribution of samples over geometrical boundaries and take into account the differences between depth and normal values at P and Q. These functions are based on the previous work of Schied et al. [14] with the depth weight w_z being

$$w_z = \exp\left(-\frac{|z(P) - z(Q)|}{\sigma_z\,|\nabla z(P)\cdot(P-Q)| + \varepsilon}\right), \tag{12}$$

where $\nabla z(P)$ is the depth gradient, $\sigma_z = 1$ is a constant value defined by experimentation, and $\varepsilon = 0.0001$ is a small constant value to prevent division by zero. In addition, the weight w_n is based on the difference between normals at P and Q and is defined as

$$w_n = \max\left(0, \mathbf{n}(P)\cdot\mathbf{n}(Q)\right)^{\sigma_n}, \tag{13}$$

where $\sigma_n = 32$ is again a constant value based on experimentation.

32.4.1.2 ROUGHNESS WEIGHT

Roughness weights simulate the effect of roughness on the reflection lobe. First, we only allow samples that have similar roughness values, and thus a similar shape of reflection lobe, compared to the current pixel:

$$w_r = f_w\left(r_{near}, r_{far}, \left|r(P) - r(Q)\right|\right), \tag{14}$$

where $r_{\text{near}} = 0.01$ and $r_{\text{far}} = 0.1$ are constants chosen based on experimentation. Second, we control the filtering radius for the contributing samples based on roughness with weight

$$w_d = f_w\left(d_{\text{near}}, d_{\text{far}}, \|\mathbf{d}\|\right), \tag{15}$$

where $d_{\text{near}} = 10\, r(P)$, $d_{\text{far}} = 70\, r(P)$, and $\mathbf{d} = P - Q$ is the vector from the current pixel position to the sample pixel position.

32.4.1.3 REFLECTION-DIRECTION WEIGHT

A *reflection-direction weight* makes the filtering kernel anisotropic by scaling it into the direction of the reflection:

$$w_s = \text{saturate}\left(\hat{\mathbf{d}}\cdot\hat{\mathbf{r}}\right)s_{c_s} + s_{c_b}, \tag{16}$$

where r is the reflection direction in screen space, $s_{c_s} = 0.5$ is a scaling factor, and $s_{c_b} = 0.5$ is a scaling bias.

32.4.1.4 RAY-LENGTH WEIGHT

The *ray-length weight* is designed to control the gathering radius as a function of the ray length: the closer the hit point is, the less we want the neighboring samples to contribute. Therefore, the weight w_l becomes

$$w_l = f_w\left(l_{\text{near}}, l_{\text{far}}, \|\mathbf{d}\|\right), \tag{17}$$

where $l_{\text{near}} = 0$ and $l_{\text{far}} = 10.0\, r(P)$.

Finally, we can combine all the weights into a single function:

$$w(P,Q) = w_z(P,Q)\, w_n(P,Q)\, w_r(P,Q)\, w_d(P,Q)\, w_s(P,Q)\, w_l(P,Q). \tag{18}$$

32.4.2 TEMPORAL FILTERING

Unfortunately, the spatial filter is often not sufficient to reach the desired quality. Therefore, in addition to accumulating samples in the pixel neighborhood, we also accumulate them temporally over multiple frames. This is done by interpolating between the current frame samples and the previous temporal results using an exponential moving average:

$$C_i = (1-\gamma)\, S_i + \gamma\, C_{i-1}, \tag{19}$$

where C_i is the current frame output, C_{i-1} is the previous frame output projected using a velocity vector, and S_i is the current frame input (i.e., the reflection buffer). Acquiring these velocity vectors for reflections is discussed in further detail in Section 32.4.3. The weight γ denotes the ratio of interpolation between the history data and the current frame and is based on multiple heuristics.

Glossy reflection can have significant color variance between temporal samples. This prevents us from relying on methods based on color values, such as variance clipping, to remove ghosting. Instead, we use a subset of the geometry-based weight functions from Section 32.4.1 to define γ. This is done by first projecting P to generate the sampling location of the contributing sample using a velocity vector and next using that to sample the surface attributes of the previous frame. Thus, we must also save the depth and normal attributes from the G-buffer of the previous frame.

In addition, we include a weight $w_{r_{\max}}$ that is based on the roughness of the current sample. This is done so that extremely smooth surfaces, such as mirrors, disregard unnecessary temporal samples to remove any possible ghosting. This weight is calculated as follows:

$$w_{r_{\max}} = \text{smoothstep}\left(0, r_{\max}, r\left(P\right)\right), \tag{20}$$

where $r_{\max} = 0.1$ is a constant threshold. Therefore,

$$\gamma = 0.95\, w_z\left(P, Q\right) w_n\left(P, Q\right) w_r\left(P, Q\right) w_{r_{\max}} \tag{21}$$

is the total weight used to weight the current and the previous frames.

Nevertheless, ghosting can still appear on planar surfaces with a constant roughness that is large enough not to be clamped by $w_{r_{\max}}$ but smooth enough for ghosting to be clearly visible. These artifacts are most noticeable with reflections of bright light sources or quickly moving brightly colored objects. Unfortunately, this cannot be solved by geometry weight functions because we cannot account for differences between the objects visible in reflection by comparing the reflector surfaces. Thus, we choose to implement a 5 × 5 filtering kernel for the current reflection result to calculate variance for both incoming light L and the filtered BRDF while using edge-stopping functions to prevent sampling over geometrical boundaries. These are then used for color-space variance clipping of the temporal filtering result C_i, and thus they prevent blending of the temporal results with completely different color values compared to the current frame. This is similar to variance clipping commonly done with temporal antialiasing, only with nonuniform sample weights to prevent sampling over geometrical boundaries.

32.4.3 REFLECTION MOTION VECTORS

Motion vectors need to be adjusted for objects seen through a reflection as the velocity is not just a projection of the object's velocity onto the screen.

32.4.3.1 UNDERSTANDING THE PROBLEM

To tackle this problem we will start with a fully static system: a camera, a reflector, and an object that is seen in the reflection. In Figure 32-7, light emanates from the object at P_o in multiple directions; one of the photons perfectly reflects from the reflector at $P_{s,0}$ and reaches the eye. The object is detected as if it were somewhere along the ray that hit the eye.

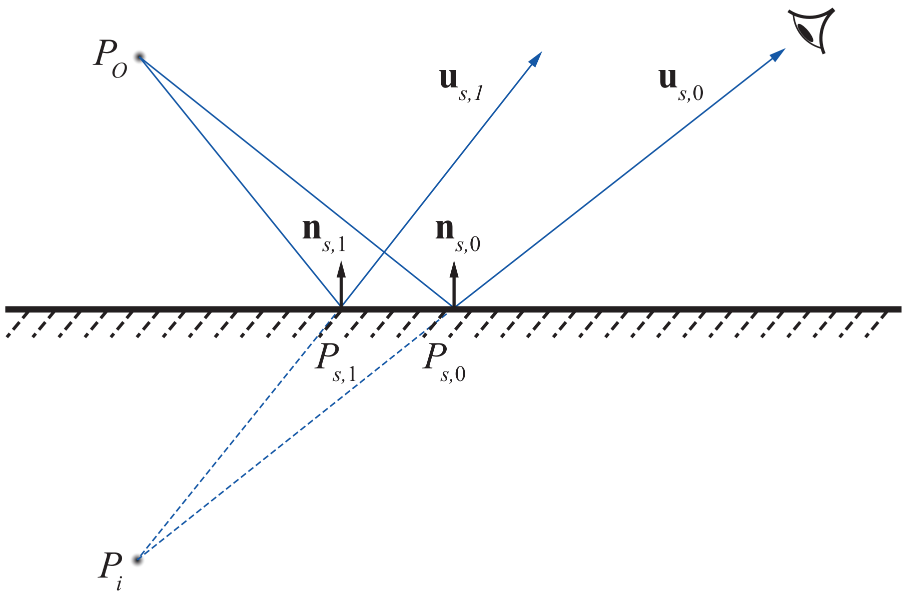


Figure 32-7. *Mirror reflections in a plane.*

Note that if the eye moved to another location, to which the light from the object reaches as well, then that object would appear at the same place. Moving the eye around to new points, we get multiple rays that intersect somewhere beneath the surface. That intersection point is P_i, and it is called the *image* of the object. In this particular case, the rays intersect in their *past*, so the image is virtual, while for other configurations rays can intersect in their *future* and we get a real image. Furthermore, in real configurations, rays often do not intersect perfectly, and instead a circle of confusion is obtained. However, for the purposes of solving this problem we are going to ignore this scenario and assume that the rays always converge.

The general strategy should now become clear. We want to replace explicit treatment of the object and trying to collect its velocity by treatment of the object's image and using its velocity as is. It is also more natural to treat the image rather than the object, because what we see on the screen is the image, so we should be analyzing it and not the object itself.

32.4.3.2 DIRECT SOLUTION

A straightforward way of finding P_i, in the sense of least squares, is the solution to the intersection of lines as given by

$$\begin{aligned} \left(\mathbf{I} - \sum_j \mathbf{u}_{s,j}\mathbf{u}_{s,j}^\mathsf{T}\right) P_i &= \sum_j \left(\mathbf{I} - \mathbf{u}_{s,j}\mathbf{u}_{s,j}^\mathsf{T}\right) P_{s,j}, \\ \text{where} \quad \mathbf{u}_{s,j} &= \left(2\mathbf{n}_{s,j}\mathbf{n}_{s,j}^\mathsf{T} - \mathbf{I}\right)\left(P_o - P_{s,j}\right). \end{aligned} \tag{22}$$

Here, $P_{s,j}$ are the points on the surface in a local footprint, and $\mathbf{u}_{s,j}$ are the reflection directions from these points, as shown in Figure 32-7.

The solution for velocity can be obtained after differentiating with respect to time. However, this is cumbersome and requires a significant amount of information.

32.4.3.3 GEOMETRICAL OPTICS APPROACH

If we assume that the reflector point is umbilical, we can simplify the problem significantly. An *umbilical* point is locally sphere-like, and the problem of finding the image of an object reflected from a spherical surface can be solved by the thin lens equations, which are given by

$$\begin{gathered} \frac{1}{f} = -\frac{2}{r} \\ \Leftrightarrow \\ \frac{1}{f} = \frac{1}{z_o} + \frac{1}{z_i} \\ \Leftrightarrow \\ \frac{x_i}{x_o} = -\frac{z_i}{z_o}, \quad \frac{y_i}{y_o} = -\frac{z_i}{z_o}, \end{gathered} \tag{23}$$

where r is the radius of curvature.

32.4.3.4 OBTAINING OPTICAL PARAMETERS

Initial part of this section assumes that the reader is familiar with topics from differential geometry of surfaces. For a thorough discussion on differential geometry we refer the reader to do Carmo's book [4].

In Figure 32-8, P_s depicts a reflector's point in world-space coordinates, which is projected onto a pixel (seen on screen), and $P_{s,j}$ is the surface point of a neighbor pixel. The reflection interaction occurs in the *normal plane*, the plane that is orthogonal to the tangent plane at the point of reflection and contains the view vector. In that plane, the radius of the circular reflector is the inverse of the normal curvature κ_n in the view direction projected onto the tangent plane at P_s, the point of reflection. However, κ_n is dependent on the view and changes when the camera is moving. Hence, instead of using the normal curvature in the view direction, we use the principal curvature κ_s that produces the closest image to the viewer. This value can be found by calculating both principal curvatures and choosing the one that produces the nearest image in front of the viewer (negative curvatures can produces images behind the viewer). This decision effectively forces the reflector point to be umbilical (since the same normal curvature is always used, regardless of the view). Since r is the inverse of κ_s, it can be infinite (for a plane), but it cannot be zero. The same applies for the focus. Hence, we will work with inverse quantities of the radius and focus.

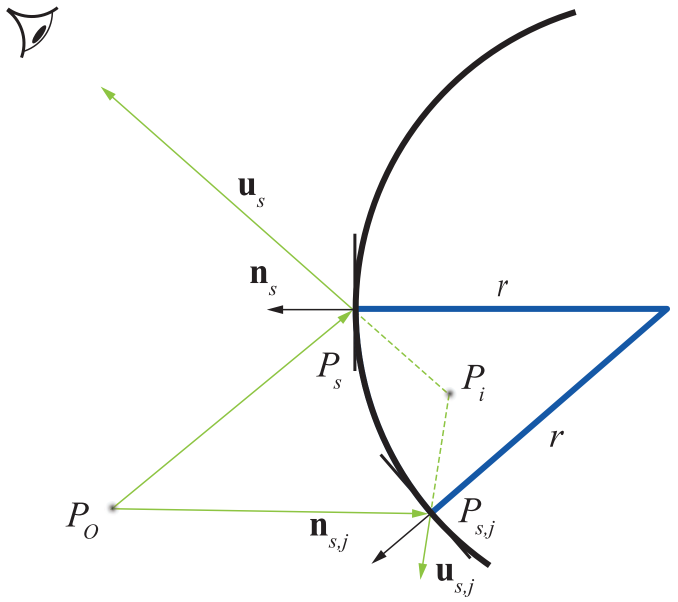


Figure 32-8. *Reflection from a spherical mirror.*

For an orthonormal basis $\{\mathbf{x}, \mathbf{y}, \mathbf{n}_s\}$, the reflected object coordinates are

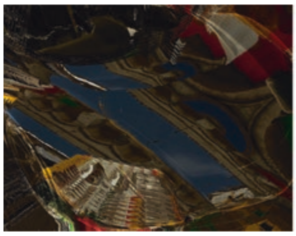
$$x_o = \mathbf{x} \cdot (P_o - P_s), \quad y_o = \mathbf{y} \cdot (P_o - P_s), \quad z_o = \mathbf{n}_s \cdot (P_o - P_s), \tag{24}$$

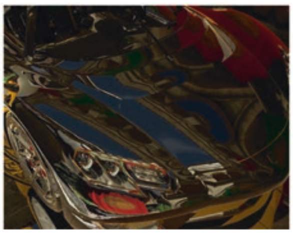
where P_o and P_s are the world-space positions of the reflected object and the reflector, respectively. The image coordinates $(x_i \;\; y_i \;\; z_i)^\mathsf{T}$, which are obtained from the thin lens equations, specify the image position in world space as

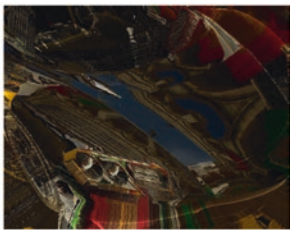
$$P_i = P_s + x_i\,\mathbf{x} + y_i\,\mathbf{y} + z_i\,\mathbf{n}_s. \tag{25}$$

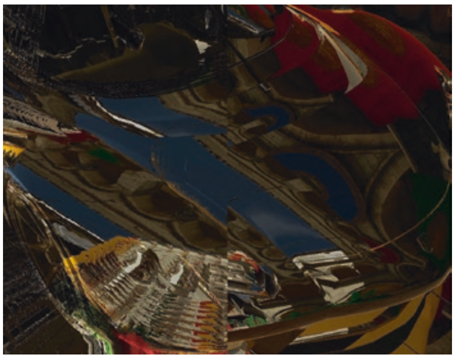
Note that when this is used as input for the temporal filtering pass of glossy reflections, we want to extend the sample count temporally, i.e., find a sample from the history that was likely sampled from a similar distribution instead of a sample that likely intersected the same position. Therefore, we use an estimate of the intersection of a most likely ray by using the length of the highest-probability ray of the pixel multiplied by the primary reflection direction.

Motion vectors computed with our approach provide better estimates for reprojecting hit positions in curved surfaces than primary hit surface motion vectors, or than approaches that do not take the curvature of the reflecting surface into account. This is shown in Figure 32-9.


(a) Planar.


(b) Reference.


(c) Thin lens.


(d) Planar (left) versus reference (right). Notice that the reflection on the left appears as if the camera was not zoomed out at all.

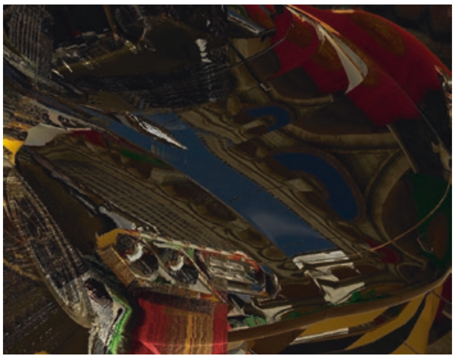

(e) Thin lens (left) versus reference (right). Note that the boundary in the middle closely matches between the images.

Figure 32-9. *Comparison between view-dependent velocity vectors with the assumed planar reflector and the thin lens approximation. The camera is zoomed out approximately 1 meter during 10 frames while sampling only the previous frame's data from the screen, using coordinates offset with the motion vectors. Reflections are calculated only for the first frame.*

32.4.3.5 VELOCITY TRANSFORMATION FOR DYNAMIC OBJECTS

If the basis vectors x and y were selected without dependency on the view, then they don't have time dependency with respect to the camera position. They do have time dependency with respect to the surface normal, which changes when the reflector rotates. However, we will neglect this change and assume $\dot{\mathbf{x}} = 0$, $\dot{\mathbf{y}} = 0$, and $\dot{\mathbf{n}} = 0$. The temporal derivatives of Equation 24 and Equation 25 are then

$$\dot{x}_o = \mathbf{x} \cdot \left(\dot{P}_o - \dot{P}_s\right), \quad \dot{y}_o = \mathbf{y} \cdot \left(\dot{P}_o - \dot{P}_s\right), \quad \dot{z}_o = \mathbf{n}_s \cdot \left(\dot{P}_o - \dot{P}_s\right), \\ \text{where} \quad \dot{P}_i = \dot{P}_s + \dot{x}_i \mathbf{x} + \dot{y}_i \mathbf{y} + \dot{z}_i \mathbf{n}_s. \tag{26}$$

From Equation 23, we obtain

$$\dot{z}_i = \left(\frac{z_i}{z_o}\right)^2 \dot{z}_o,$$
$$\dot{x}_i = -\frac{z_i}{z_o}\dot{x}_o + \frac{x_o}{f}\left(\frac{z_i}{z_o}\right)^2 \dot{z}_o, \tag{27}$$
$$\dot{y}_i = -\frac{z_i}{z_o}\dot{y}_o + \frac{y_o}{f}\left(\frac{z_i}{z_o}\right)^2 \dot{z}_o.$$

The velocity of the image point can be readily calculated from Equations 26 and 27, then projected to the screen to obtain the screen-space motion vectors as follows. Given the matrix **M** that transforms from world coordinates to screen coordinates and the screen coordinates (x_{ss}, y_{ss}) of the reflector,

$$v = \frac{1}{\mathbf{m}_{,3} \cdot p_i}\left(\begin{pmatrix} \mathbf{m}_{,0} \cdot \dot{P}_i \\ \mathbf{m}_{,1} \cdot \dot{P}_i \end{pmatrix} - \begin{pmatrix} x_{ss} \\ y_{ss} \end{pmatrix} \cdot \left(\mathbf{m}_{,3} \cdot \dot{P}_i\right)\right), \tag{28}$$

where $\mathbf{m}_{,i}$ denotes the *i*th row of **M**. This accounts only for the velocity of the image; the additional velocity component caused by camera movement needs to be added separately. However, since velocity is relative, the camera's velocity can be subtracted from both the object's and reflector's velocities in Equation 26. This makes the matrix **M** independent of time for this calculation. Another method to calculate the screen-space motion vector is by advancing the image position backward in time with an Euler iteration, projecting it to the screen, and taking the difference in screen space.

32.5 RESULTS

We measured the results of our algorithm in the standard Sponza scene in five different scenarios. We compare against a fully shaded reference, i.e., specular computations without the cache. The scene was fitted with 11 cache sampling points. We used a roughness threshold (RT_MAX_ROUGHNESS) of 0.8. All numbers were captured on an NVIDIA RTX 2080 GPU at a resolution of 2560 × 1440.

In addition to the final illumination and the reflection term images shown in Figure 32-10, we also include images of our *reflection mask*. The mask is a color-coded visualization of the type of reflection path per pixel. Purple color in the mask denotes the cheapest path: sampling with just a direction vector. Green and orange areas are ray traced: radiance is sampled from the screen space for dark green pixels, from the cache for the light green pixels, and fully computed for the orange pixels, denoting the most expensive computation path.

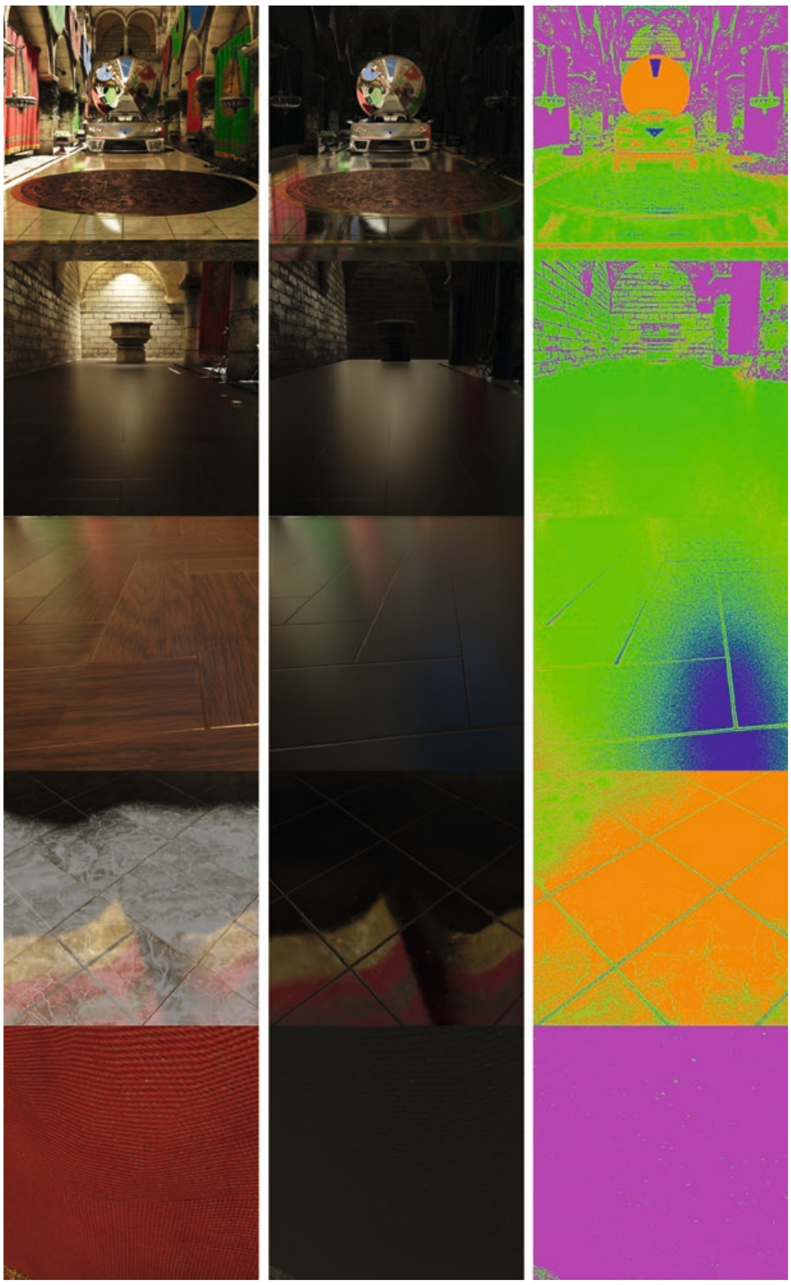


Figure 32-10. *Test cases from top to bottom: Main, Spot, Wood, Tile, and Curtain.Left: the final illumination. Center: the environment reflection term. Right: the reflection mask (color map as described in Section 32.5). These images were captured from a static camera for a static scene.*

32.5.1 PERFORMANCE

We measured the performance with sample counts of one and one to four, scaled with surface roughness. The results are shown in Tables 32-1 and 32-2, respectively. Performance is given for each pass separately. Rays are traced only in the ray tracing pass, which writes out all necessary data for the possible ray shading pass. Cache relighting time was (naturally) constant for all the test cases. The performance of the other parts depends mostly on the number of samples taken and the utilization rate of the radiance cache. If the cache cannot be used at all, our technique reverts to full shading of the rays. In this case the overhead from cache illumination and sampling (all samples rejected) is paid in full in addition to the cost of full ray shading. The Tile scenario covers such a case in which our algorithm performs similarly as full shading.

Table 32-1. *Performance of various passes on an NVIDIA RTX 2080 for different cameras in frame time (ms) when a single sample is taken per pixel. Our technique is denoted with "(o)" and the fully shaded comparison with "(f)." The numbers were captured in the Sponza scene. In all cases filtering took approximately 10 ms. Images matching these test cases can be seen in Figure 32-10.*

	Ray Tracing	Cache Relighting	Cache Sampling +Velocity +Work Generation	Ray Shading	Total
Main (o) Main (f)	2.68 3.62	0.87 0	1.70 0.66	1.25 8.40	5.63 12.68 (2.25x)
Spot (o) Spot (f)	2.08 2.39	0.88 0	0.89 0.63	0.62 6.41	3.59 9.43 (2.63x)
Wood (o) Wood (f)	3.87 3.84	0.88 0	1.04 0.62	1 7.06	5.91 11.52 (1.95x)
Tile (o) Tile (f)	3.19 3.27	0.88 0	0.91 0.59	3.60 4.62	7.70 8.48 (1.10x)
Curtain (o) Curtain (f)	0.24 3.10	0.88 0	1.10 0.66	0.33 8.81	1.72 12.57 (7.31x)

Table 32-2. *Same as in Table 32-1 but with one to four samples. Sample counts were dynamically selected for each pixel based on roughness (increase sample count as the surface gets rougher).*

	Ray Tracing	Cache Relighting	Cache Sampling +Velocity +Work Generation	Ray Shading	Total
Main (o) Main (f)	8.16 12.45	0.88 0	5.24 1.02	3.99 34.13	17.39 47.60 (2.74x)
Spot (o) Spot (f)	8.16 9.57	0.88 0	4.64 1.09	1.63 28.24	14.43 38.90 (2.70x)
Wood (o) Wood (f)	15.50 15.45	0.88 0	3.32 1.07	3.96 35.47	22.78 51.99 (2.28x)
Tile (o) Tile (f)	8.51 8.61	0.88 0	2.49 0.95	8.97 12.29	19.97 21.85 (1.09x)
Curtain (o) Curtain (f)	0.51 12.78	0.88 0	3.74 1.04	0.4 39.98	4.65 53.80 (11.57x)

Our algorithm has highest performance when the ray tracing part can be skipped completely. This can be seen in the Curtain scenario with a rough material. In this case the performance difference is almost 7× with one sample and 15× with multiple samples.

Scenarios Spot and Wood sample from either screen space or the radiance cache. These scenarios require ray tracing but still take the fast path during shading. In these cases our algorithm is approximately 2× faster than full shading. Reflections in these cases are glossy, which helps our cache use.

A balanced example can be seen in the Main scenario. This shot contains all types of surfaces from rough rocks to polished tile floors. Again, we measure 2.5× performance improvement compared to full shading.

32.5.2 QUALITY

Figure 32-11 shows a smooth surface with reflections computed using our technique compared to a per-ray shaded reference. The quality of our technique is comparable even though some of the samples are fetched from the low-resolution cache. In general, the quality of reflections does not greatly depend on the cache sizes due to our sampling heuristics. Smaller caches will result in more misses, but the overall quality stays close to the reference. This is shown in Figure 32-12 where a cache resolution of 256 × 256 is compared against 32 × 32.

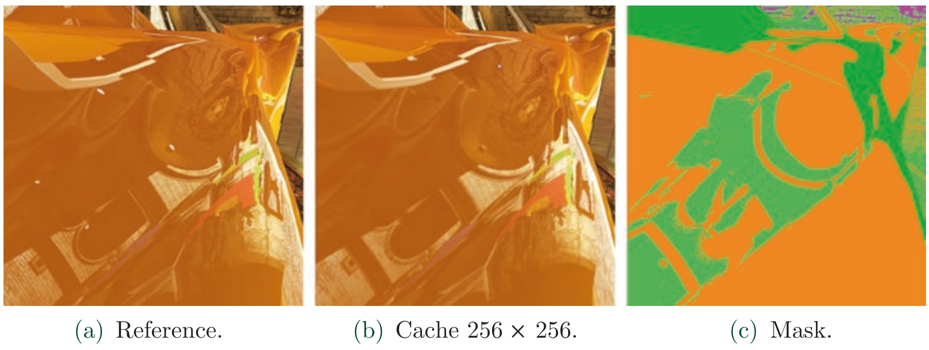


Figure 32-11. *Reference compared to our technique when α = 0, i.e., the material is mirror-like, including the reflection mask. Some parts of the surface still sample from the cache due to our sampling heuristics.*

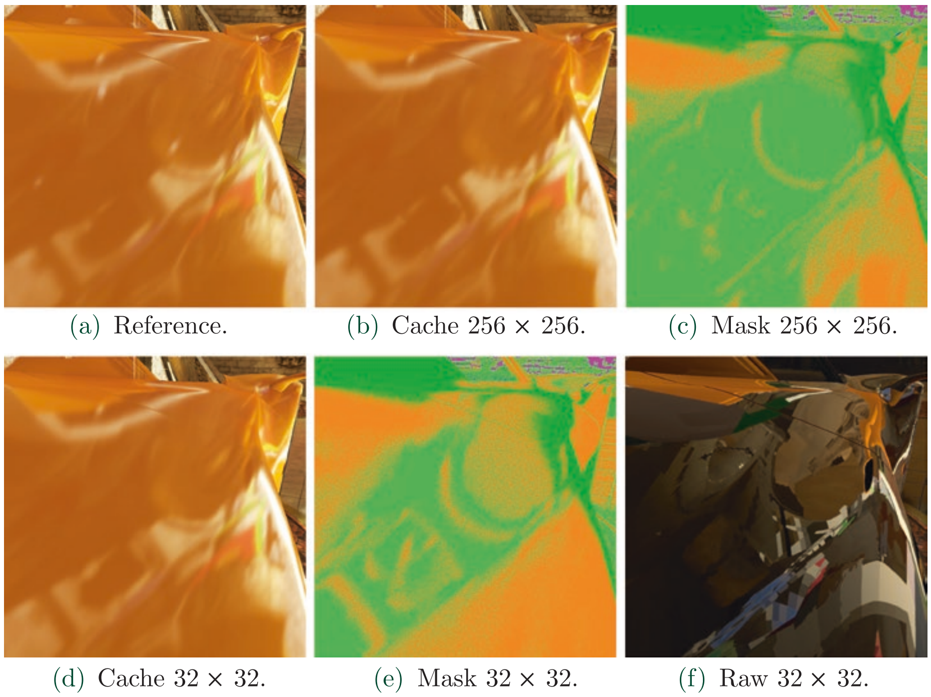
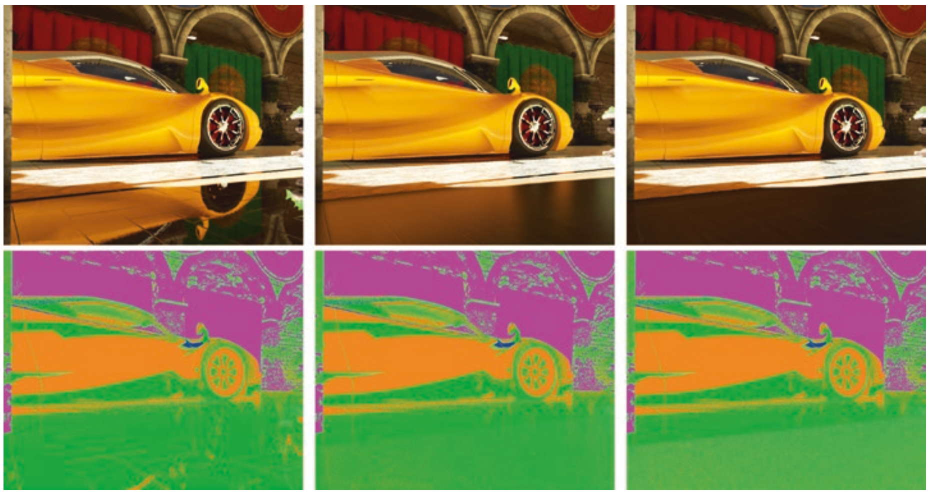


Figure 32-12. *Reference compared to our technique when α = 0.1 with two different cache sizes. Note how cache hits are greatly increased by our sampling heuristics compared to Figure 32-11. Even cache size 32 × 32 produces lots of cache hits for rougher surfaces but naturally less than size 256 × 256. The last image shows reflection sampling from the cache with heuristics disabled.*

As the roughness of the surface increases, the noise naturally increases as well, notably when a single sample per pixel is used. However, the spatiotemporal filtering can greatly reduce this noise, and multiple samples may be taken to balance the cost of the filtering. With rougher surfaces, the limited-resolution radiance caches are more effective, as shown in Figure 32-13, which makes the multiple samples approach more affordable with our technique. Having multiple samples is also cheaper using our technique because reuse of the radiance cache increases and only the cache misses will have to be shaded redundantly.



(a) $\alpha = 0.0.$ **(b)** $\alpha = 0.3.$ **(c)** $\alpha = 0.6.$

Figure 32-13. *Varying the material roughness (α) of the floor, with reflections maps at the bottom. As the material gets rougher, more samples are fetched from the cache or fully shaded as they deviate from screen space: this can be seen at the bottom of the mask as the color turns from dark green to light green and orange. Mirror-like surfaces sample effectively from screen space when possible.*

The noise reduction of the spatiotemporal filtering is shown in more detail in Figure 32-14. While variance clipping cannot remove all ghosting caused by moving silhouettes in reflections and thus leaves small artifacts, these are harder to notice when the camera is moving. Also, the roughness of the curtain on the right is above the RT_ROUGHNESS_THRESHOLD and radiance is inaccurately sampled from the other side, but this issue, although often difficult to notice in the final result, could be alleviated by more careful probe placement. Apart from the mentioned artifacts, the overall result is close to the reference image, which is computed with multiple samples until convergence.

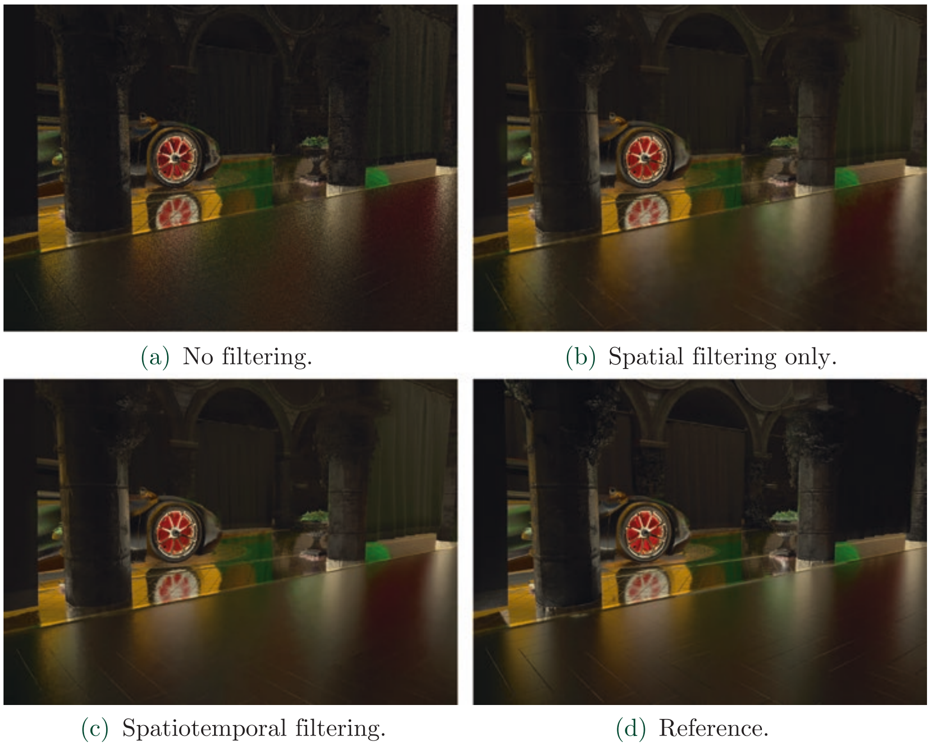


Figure 32-14. *Effects of filtering on the raw specular illumination term. The camera is moving rightward approximately 5 m/s, the car is moving rightward approximately 0.8 m/s, and the frame rate is 30 Hz.*

32.6 CONCLUSION

In this chapter, we have presented a technique for producing realistic real-time specular illumination for dynamic scenes. Our approach combines old and new techniques: we use the new DXR API for querying scene visibility, but do most of the shading in either screen space or cache space. In both of these cases, the efficiency of modern GPUs is well utilized due to coherency between neighboring threads. Only some rays go through the more costly, divergent full-shading path. Immense performance improvements can be measured especially for rougher surfaces that go over the roughness threshold: for these surfaces the ray cast can be completely skipped, thus eliminating many rays. However, even without ray tracing, these surfaces get a real-time specular term from our constantly updated sparse-lighting cache.

32.7 FUTURE WORK

There are avenues for improvement in various parts of the algorithm:

> *Indirect diffuse:* A similar approach can be used to compute indirect diffuse lighting. Rays that miss the cache can get the information from a low-frequency source, such as a hole-filling algorithm.

> *Improved cache illumination:* Our cache is at the moment illuminated each frame. However, an improved system could be built that only illuminates those cubes, faces, or even samples that are actually used. For example, only the most important cubes could be lit per frame.

> *Radiance cache geometry:* The implementation described here uses cube maps, i.e., spherical captures, for cache storage. However, this wastes space as the same surfaces can be seen by different cache points. Therefore, we plan to investigate other cache data structures for an improved cache utilization.

> *Hole filling:* A reflection mask can be very noisy for some surfaces, meaning that some neighboring pixels either sample from cubes or shade the full ray. As shading the full ray is more costly, some of the small holes could be filled based on the neighboring pixel data, especially for rougher surfaces.

> *Filtering:* The filter presented in this chapter is somewhat expensive for real-time use. In the future we aim to look for lighter filtering solutions that make different trade-offs between quality and performance.

REFERENCES

[1] Crassin, C., Neyret, F., Sainz, M., Green, S., and Eisemann, E. Interactive Indirect Illumination Using Voxel Cone Tracing. *Computer Graphics Forum 30*, 7 (2011), 1921–1930.

[2] 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.

[3] Debevec, P. Image-Based Lighting. HDRI and Image-Based Lighting, SIGGRAPH Courses, August 2003.

[4] do Carmo, M. P. *Differential Geometry of Curves and Surfaces*. Prentice Hall Inc., 1976.

[5] Elcott, S., Chang, K., Miyamoto, M., and Metaaphanon, N. Rendering Techniques of Final Fantasy XV. In *SIGGRAPH Talks* (2016), pp. 48:1–48:2.

[6] Halton, J. H. Algorithm 247: Radical-Inverse Quasi-Random Point Sequence. *Communications of the ACM 7*, 12 (1964), 701–702.

[7] Heitz, E. A Simpler and Exact Sampling Routine for the GGX Distribution of Visible Normals. Research report, Unity Technologies, Apr. 2017.

[8] Heitz, E., Dupuy, J., Hill, S., and Neubelt, D. Real-Time Polygonal-Light Shading with Linearly Transformed Cosines. *ACM Transactions on Graphics 35*, 4 (July 2016), 41:1–41:8.

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

[10] Karis, B. Real Shading in Unreal Engine 4. Physically Based Shading in Theory and Practice, SIGGRAPH Courses, August 2013.

[11] Lagarde, S., and Zanuttini, A. Local Image-Based Lighting with Parallax-Corrected Cubemaps. In *SIGGRAPH Talks* (2012), p. 36:1.

[12] McGuire, M., and Mara, M. Efficient GPU Screen-Space Ray Tracing. *Journal of Computer Graphics Techniques 3*, 4 (December 2014), 73–85.

[13] McGuire, M., Mara, M., Nowrouzezahrai, D., and Luebke, D. Real-Time Global Illumination Using Precomputed Light Field Probes. In *Symposium on Interactive 3D Graphics and Games* (2017), pp. 2:1–2:11.

[14] 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.

[15] Schlick, C. An Inexpensive BRDF Model for Physically-based Rendering. *Computer Graphics Forum 13*, 3 (1994), 233–246.

[16] Stachowiak, T. Stochastic Screen-Space Reflections. Advances in Real-Time Rendering in Games, SIGGRAPH Courses, August 2015.

[17] Stachowiak, T. Stochastic All the Things: Raytracing in Hybrid Real-Time Rendering. Digital Dragons Presentation, 2018.