Chapter 25 in Ray Tracing Gems, edited by Eric Haines and Tomas Akenine-Möller, Apress, 2019.
License: Creative Commons Attribution 4.0 International License (CC-BY-NC-ND), https://raytracinggems.com

Hybrid Rendering for Real-Time Ray Tracing

Colin Barré-Brisebois, Henrik Halén, Graham Wihlidal, Andrew Lauritzen,
Jasper Bekkers, Tomasz Stachowiak, and Johan Andersson
SEED / Electronic Arts

Abstract

This chapter describes the rendering pipeline developed for *PICA PICA*, a real-time ray tracing experiment featuring self-learning agents in a procedurally assembled world. *PICA PICA* showcases a hybrid rendering pipeline in which rasterization, compute, and ray tracing shaders work together to enable real-time visuals approaching the quality of offline path tracing.

The design behind the various stages of such a pipeline is described, including implementation details essential to the realization of *PICA PICA*'s hybrid ray tracing techniques. Advice on implementing the various ray tracing stages is provided, supplemented by pseudocode for ray traced shadows and ambient occlusion. A replacement to exponential averaging in the form of a reactive multi-scale mean estimator is also included. Even though *PICA PICA*'s world is lightly textured and small, this chapter describes the necessary building blocks of a hybrid rendering pipeline that could then be specialized for any AAA game. Ultimately, this chapter provides the reader with an overall good design to augment existing physically based deferred rendering pipelines with ray tracing, in a modular fashion that is compatible across visual styles.

1 Hybrid Rendering Pipeline Overview

PICA PICA [2, 3] features a hybrid rendering pipeline that relies on the rasterization and compute stages of the modern graphics pipeline, as well as the recently added ray tracing stage [23]. See Figure 1. The reader can see such results via a video available online [10]. Visualized as blocks in Figure 2, several aspects of such a pipeline are realized by a mix-and-match of the available graphical stages, and the pipeline takes advantage of each stage's unique capabilities in a hybrid fashion.

By relying on the interaction of multiple graphical stages, and by using each stage's unique capabilities to solve the task at hand, modularization of the rendering process allows for achieving each visual aspect optimally. The interoperability of DirectX also allows for shared intermediary results between passes and, ultimately, the combination of those techniques into the final rendered image. Moreover, a compartmentalized approach as such is scalable, where techniques mentioned in Figure 2 can be adapted depending on the user's hardware capabilities. For example, primary visibility and shadows can be rasterized or ray traced, and reflections and

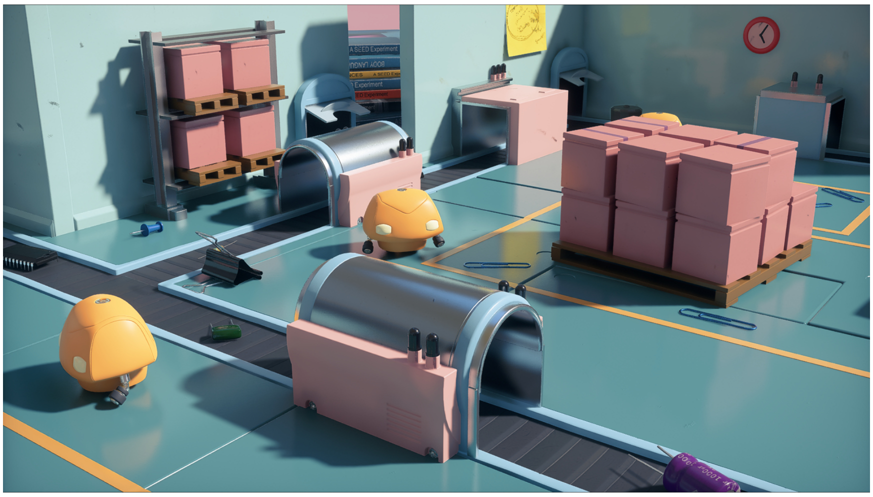


Figure 1: Hybrid ray tracing in *PICA PICA*.

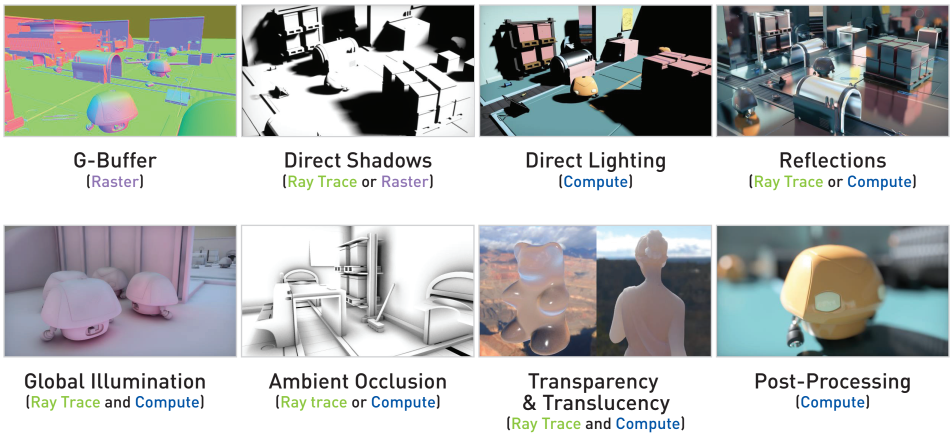


Figure 2: Hybrid rendering pipeline.

ambient occlusion can be ray traced or ray marched. Global illumination, transparency, and translucency are the only features of *PICA PICA*'s pipeline that fully require ray tracing. The various stages described in Figure 2 are executed in the following order:

1. Object-space rendering.
   1.1. Texture-space object parameterization.
   1.2. Transparency and translucency ray tracing.
2. Global illumination (diffuse interreflections).
3. G-buffer layout.
4. Direct shadows.
   4.1. Shadows from G-buffer.
   4.2. Shadow denoising.
5. Reflections.
   5.1. Reflections from G-buffer.
   5.2. Ray traced shadows at reflection intersections.
   5.3. Reflection denoising.
6. Direct lighting.
7. Reflection and radiosity merge.
8. Post-processing.

2 Pipeline Breakdown

In the following subsection we break down and discuss the rendering blocks that showcase the hybrid nature of *PICA PICA*'s pipeline. We focus on shadows, reflections, ambient occlusion, transparency, translucency, and global illumination. We will not discuss the G-buffer and the post-processing blocks, since those are built on well-documented [1] state-of-the-art approaches.

2.1 Shadows

Accurate shadowing undeniably improves the quality of a rendered image. As seen in Figure 3, ray traced shadows are great because they perfectly ground objects in a scene, handling both small- and large-scale shadowing at once.

Implementing ray traced shadows in their simplest (hard) form is straightforward: launch a ray from the surface toward the light, and if the ray hits a mesh, the surface is in shadow. Our approach is hybrid because it relies on a depth buffer generated during G-buffer rasterization to reconstruct the surface's world-space position. This position serves as the origin for the shadow ray.

Soft penumbra shadows with contact hardening are implemented by launching rays in the shape of a cone, as described in the literature [1, 21]. Soft shadows are

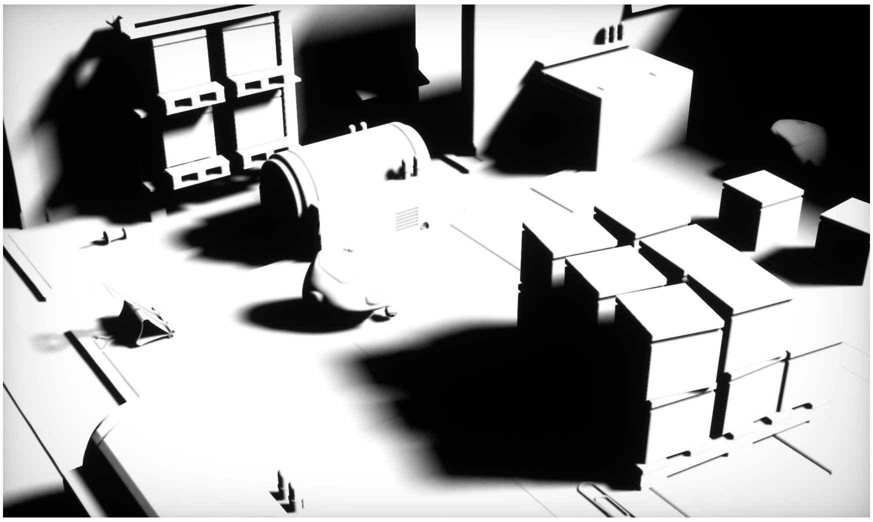


Figure 3: Hybrid ray traced soft shadows.

superior to hard shadows at conveying scale and distance, and they are also more representative of real-world shadowing. Both hard and soft shadows are demonstrated in Figure 4.

With DirectX Raytracing (DXR), ray traced shadows can be achieved by both a *ray generation shader* and *miss shader*:

```
// HLSL pseudocode---does not compile.
[shader("raygeneration")]
void shadowRaygen()
{
  uint2 launchIndex = DispatchRaysIndex();
  uint2 launchDim = DispatchRaysDimensions();
  uint2 pixelPos = launchIndex +
      uint2(g_pass.launchOffsetX, g_pass.launchOffsetY);
  const float depth = g_depth[pixelPos];

  // Skip sky pixels.
  if (depth == 0.0)
  {
    g_output[pixelPos] = float4(0, 0, 0, 0);
    return;
  }

  // Compute position from depth buffer.
  float2 uvPos = (pixelPos + 0.5) * g_raytracing.viewDimensions.zw;
  float4 csPos = float4(uvToCs(uvPos), depth, 1);
  float4 wsPos = mul(g_raytracing.clipToWorld, csPos);
  float3 position = wsPos.xyz / wsPos.w;

  // Initialize the Halton sequence.
  HaltonState hState =
      haltonInit(hState, pixelPos, g_raytracing.frameIndex);

  // Generate random numbers to rotate the Halton sequence.
  uint frameseed =
```

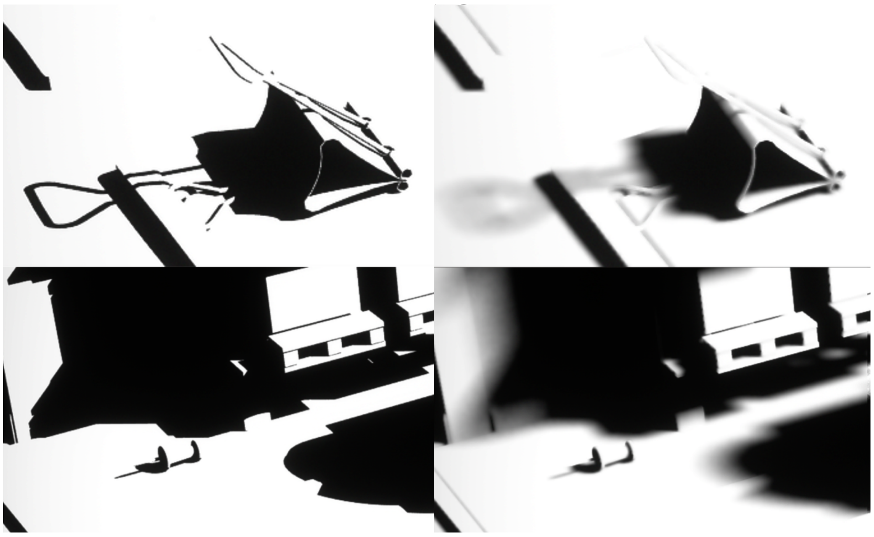


Figure 4: Hybrid ray traced shadows: hard (left) and soft and filtered (right).

```
        randomInit(pixelPos, launchDim.x, g_raytracing.frameIndex);
    float rnd1 = frac(haltonNext(hState) + randomNext(frameseed));
    float rnd2 = frac(haltonNext(hState) + randomNext(frameseed));

    // Generate a random direction based on the cone angle.
    // The wider the cone, the softer (and noisier) the shadows are.
    // uniformSampleCone() from [pbrt]
    float3 rndDirection = uniformSampleCone(rnd1, rnd2, cosThetaMax);

    // Prepare a shadow ray.
    RayDesc ray;
    ray.Origin = position;
    ray.Direction = g_sunLight.L;
    ray.TMin = max(1.0f, length(position)) * 1e-3f;
    ray.TMax = tmax;
    ray.Direction = mul(rndDirection, createBasis(L));

    // Initialize the payload; assume that we have hit something.
    ShadowData shadowPayload;
    shadowPayload.miss = false;

    // Launch a ray.
    // Tell the API that we are skipping hit shaders. Free performance!
    TraceRay(rtScene,
        RAY_FLAG_SKIP_CLOSEST_HIT_SHADER,
        RaytracingInstanceMaskAll, HitType_Shadow, SbtRecordStride,
        MissType_Shadow, ray, shadowPayload);

    // Read the payload. If we have missed, the shadow value is white.
    g_output[pixelPos] = shadowPayload.miss ? 1.0f : 0.0f;
}

[shader("miss")]
void shadowMiss(inout ShadowData payload : SV_RayPayload)
{
```

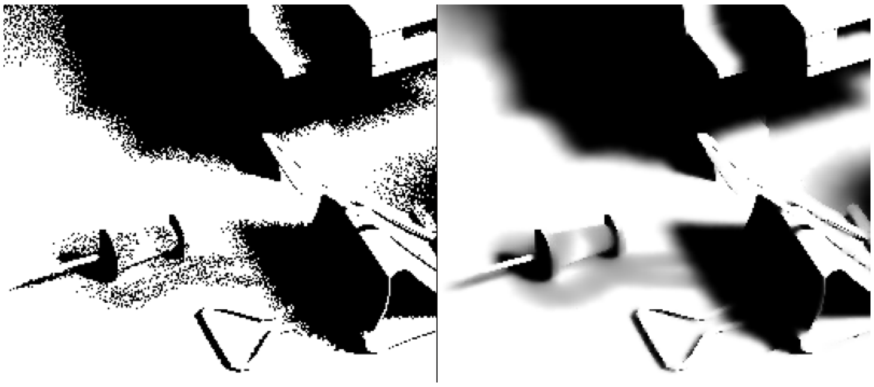


Figure 5: Hybrid ray traced shadows: unfiltered (left) and filtered (right).

```
  payload.miss = true;
}
```

As shown in this pseudocode, the miss shader payload is used to carry ray-geometry visibility information. Additionally, we use the `RAY_FLAG_SKIP_CLOSEST_HIT_SHADER` flag to inform the `TraceRay()` function that we do not require any-hit shader results. This can improve performance, since the API will know up front that hit shaders do not need to be invoked. The driver can use this information to schedule such rays accordingly, maximizing performance.

The code also demonstrates the use of the cone angle function, `uniformSampleCone()`, which drives the softness of the penumbra. The wider the angle, the softer the penumbra, but more noise will be generated. This noise can be mitigated by launching additional rays, but it can also be solved with filtering. The latter is illustrated in Figure 5.

To filter the shadows, we apply a filter derived from spatiotemporal variance-guided filtering (SVGF) [24], with a single scalar value to represent shadowing. A single scalar is faster to evaluate compared to a full color. To reduce temporal lag and improve overall responsiveness, we couple it with a pixel value bounding box clamp similar to the one proposed by Karis [15]. We calculate the size of the bounding box using Salvi variance-based method [22], with a kernelfootprint of 5×5 pixels. The whole process is visualized in Figure 6.

One should note that we implement shadows with closest-hit shaders. Shadows can also be implemented with any-hit shaders, and we could specify that we only care about the first unsorted hit. We did not have any alpha-tested geometry such as vegetation in *PICA PICA*, therefore any-hit shaders were not necessary for this demo.

Though our approach works for opaque shadows, it is possible to rely on a similar approach for transparent shadows [4]. Transparency is a hard problem in real-time graphics [20], especially if limited to rasterization. With ray tracing new alternatives are possible. We achieve transparent shadows by replacing the regular shadow tracing code with a recursive ray trace through transparent surfaces. Results are showcased in Figure 7.

In the context of light transport inside thick media, proper tracking [11] in real

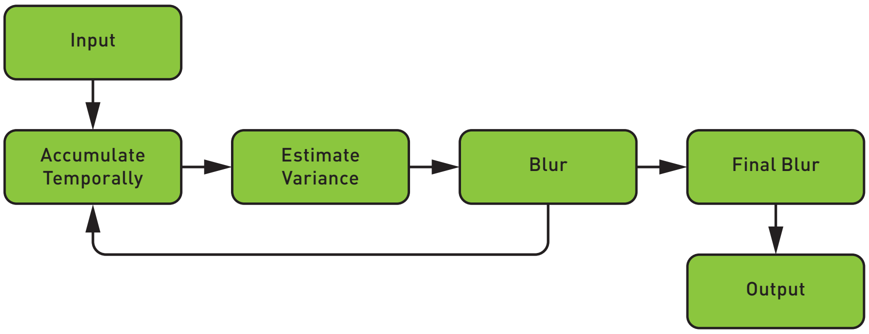


Figure 6: Shadow filtering, inspired by the work of Schied et al. [24].

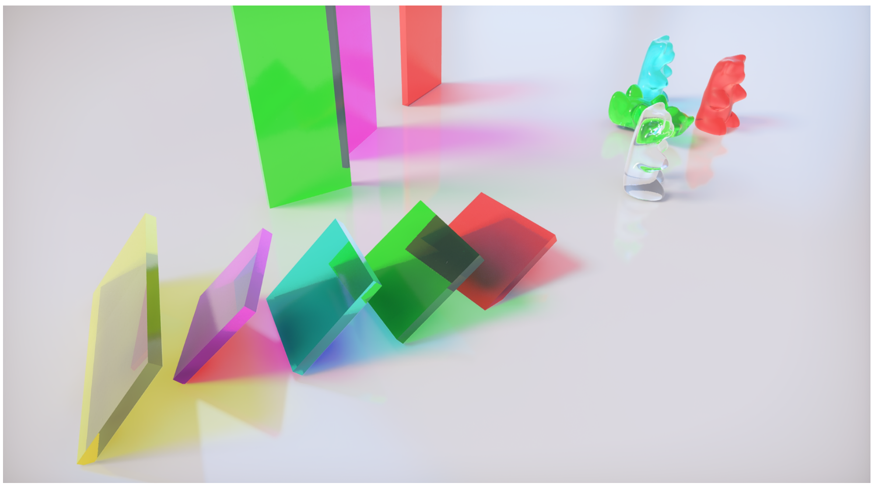


Figure 7: Hybrid ray traced transparent shadows.

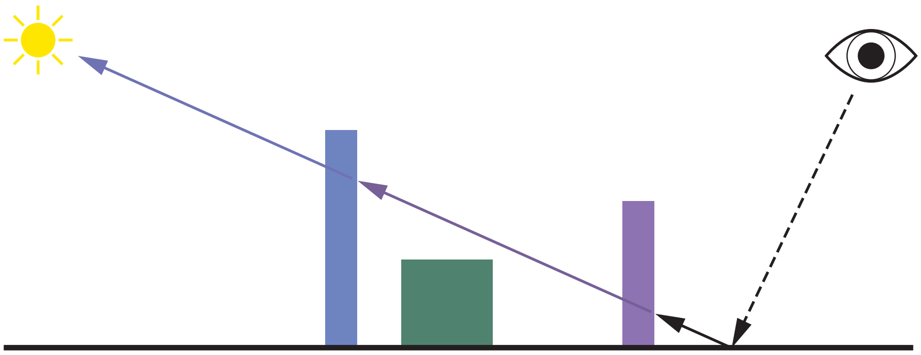


Figure 8: Hybrid ray traced transparent shadow accumulation.

time is nontrivial. For performance reasons we follow a thin-film approximation, which assumes that the color is on the surface of the objects. Implementing distance-based absorption could be a future improvement.

For any surface that needs shadowing, we shoot a ray toward the light. If we hit an opaque surface, or if we miss, we terminate the ray. However, if we hit a transparent surface, we accumulate absorption based on the albedo of the object. We keep tracing toward the light until all light is absorbed, the trace misses, or we hit an opaque surface. See Figure 8.

Our approach ignores the complexity of caustic effects, though we do take the Fresnel effect into account on interface transitions. To that, Schlick's Fresnel approximation [25] falls apart when the index of refraction on the incident side of the medium is higher than the far side. Therefore, we use a modified total internal reflection modification [16] of Schlick's model.

Similar to opaque ray traced soft shadows, we filter transparent soft shadows with our modified SVGF filter. One should note that we only compute transparent shadows in the context of direct shadowing. In the event where any other pass requires light visibility sampling, for performance reasons we approximate such visibility by treating all surfaces as opaque.

2.2 Reflections

One of the main techniques that takes advantage of ray tracing is reflections. Reflections are an essential part of a rendered image. If done properly, reflections ground objects in the scene and significantly improve visual fidelity.

Lately, video games have relied on both local reflection volumes [17] and screen-space reflections (SSR) [27] for computing reflections with real-time constraints. While such techniques can generally provide convincing results, they are often not robust. They can easily fall apart, either by lacking view-dependent information or simply by not being able to capture the complexity of interreflections. As shown in Figure 9, ray tracing enables fully dynamic complex reflections in a robust fashion.

Similar to our approach for shadows and ambient occlusion, reflection rays are launched from the G-buffer, thus eliminating the need for ray tracing of primary visibility. Reflections are traced at half resolution, or at a quarter of a ray per pixel. While this might sound limiting, a multistage reconstruction and filtering algorithm

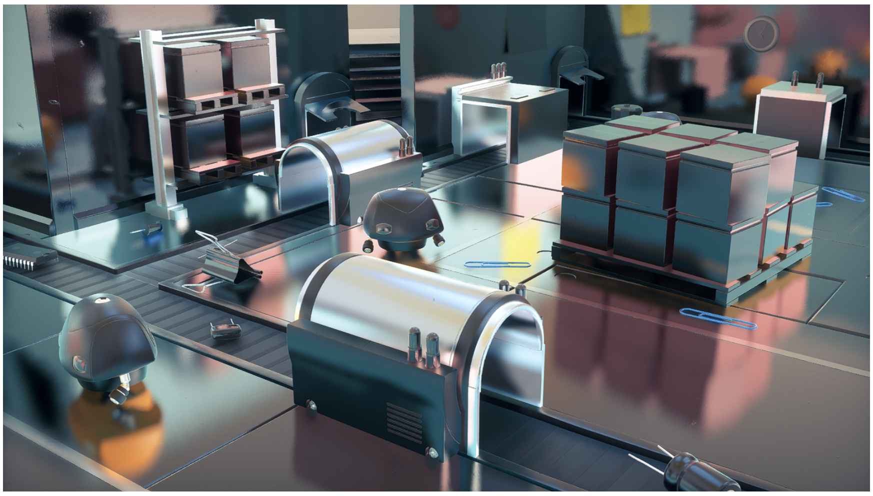


Figure 9: Hybrid ray traced reflections.

brings reflections up to full resolution. By relying on both spatial and temporal coherency, missing information can be filled and visually convincing reflections can be computed while keeping performance in check. Our technique works on arbitrary opaque surfaces, with varying normals, roughness, and material types. Our initial approach combined this with SSR for performance, but in the end we rely solely on ray traced reflections for simplicity and uniformity. Our approach relies on stochastic sampling and spatiotemporal filtering, instead of post-trace screen-space blurring. Therefore, we believe that our approach is closer to ground-truth path tracing, as surface appearance is driven by the construction of stochastic paths from the BRDF. Our approach also does not require special care at object boundaries, where blurring issues may occur with screen-space filtering approaches.

The reflection system comes with its own pipeline, as depicted in Figure 10. The process begins by generating rays via material importance sampling. Given a view direction, a reflected ray taking into account our layered BRDF is generated. Inspired by Weidlich and Wilkie's work [29], our material model combines multiple layers into a single, unified, and expressive BRDF. This model works for all lighting and rendering modes, conserves energy, and handles the Fresnel effect between layers. Sampling the complete material is complex and costly, so we only importance-sample the normal distribution. A microfacet normal is selected, which reflects the incident view vector, and a reflected ray direction is generated. As such, reflection rays follow the properties of the materials.

Since we have only a quarter of a ray per pixel, we must ensure a high-quality distribution. We use the low-discrepancy quasi-random Halton sequence because it is easy to calculate, and well distributed for low and high sample counts. We couple it with Cranley-Patterson rotation [7] for additional per-pixel jittering, in order to obtain a uniquely jittered sequence for every source pixel.

From every point in the sample space, a reflected direction is generated. Because we are sampling solely from the normal distribution, reflection rays that point below the horizon are possible. We detect this undesirable case, as depicted by the blue

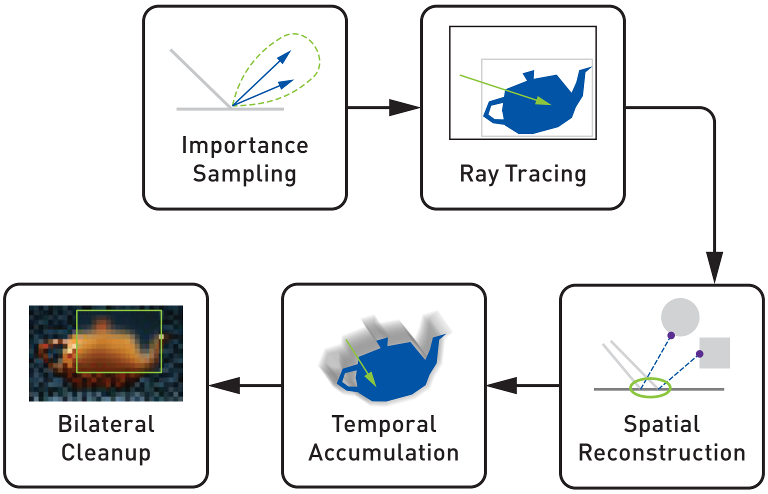


Figure 10: Reflection pipeline.

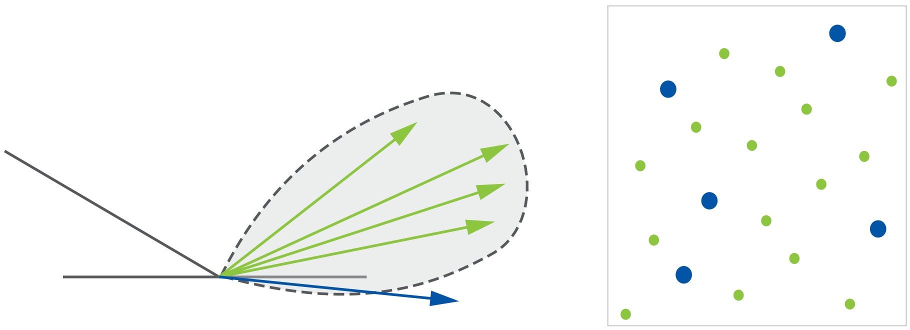


Figure 11: Left: BRDF reflection sampling. Right: Cranley-Patterson rotated Halton sequence. The probability distribution (light gray area with dashed outline) contains valid BRDF importance-sampled reflection rays (green) and reflection rays below the horizon (blue).

line in Figure 11, and compute an alternative reflection ray.

The simplest way to sample our material model is by choosing one of the layers with uniform probability and then sampling that layer's BRDF. This can be wasteful: a smooth clear coat layer is barely visible head on yet dominates at grazing angles. To improve the sampling scheme, we draw the layer from a probability mass function based on each layer's approximate visibility. See Figure 12.

After selecting the material layer, we generate a reflection ray matching its properties using the microfacet normal sampling algorithm mentioned earlier. In addition to the reflection vector, we also need the probability with which it has been sampled. We will later scale lighting contributions by the inverse of this value, as dictated by the importance sampling algorithm. It is important to keep in mind that multiple layers can potentially generate the same direction. Yet, we are interested in the probability for the entire stack, not just an individual layer. We thus

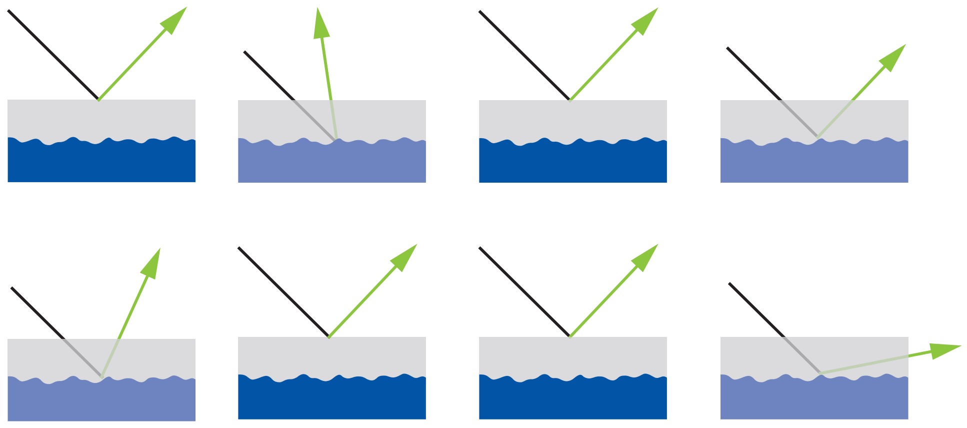


Figure 12: Eight frames of material layer sampling.

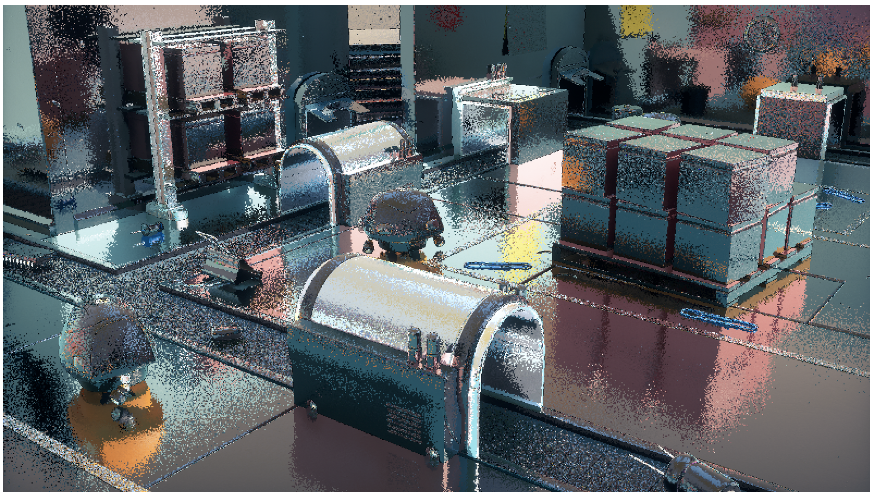


Figure 13: Hybrid ray traced reflections at a quarter ray per pixel.

add up the probabilities so that the final value corresponds to having sampled the direction from the entire stack, rather than an individual layer. Doing so simplifies the subsequent reconstruction pass, and allows using it to reason about the entire material rather than its parts.

We get results as shown in Figure 13, resembling the reflection component of the path traced image but at half resolution and with a single bounce.

```
result    = 0.0
weightSum = 0.0

for pixel in neighborhood:
    weight = localBrdf(pixel.hit) / pixel.hitPdf
    result += color(pixel.hit) * weight
    weightSum += weight

result /= weightSum
```

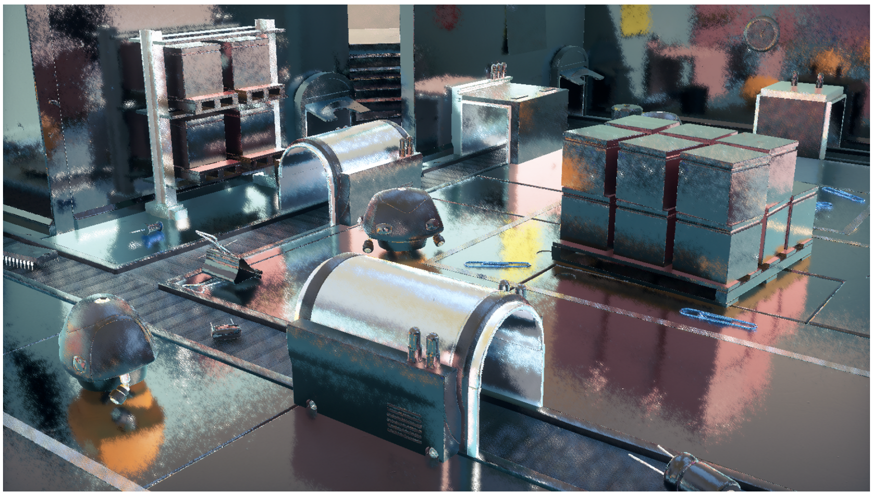


Figure 14: Hybrid ray traced reflections reconstructed at full resolution.

Once the half-resolution results have been computed, the spatial filter is applied. Results are shown in Figure 14. While the output is still noisy, it is now full resolution. This filter also gives variance reduction similar to actually shooting 16 rays per pixel, similar to work by Stachowiak [27] and Heitz et al. [12]. Every full-resolution pixel uses a set of ray hits to reconstruct its reflection, and there is a weighted average where the local pixel's BRDF is used to weigh contributions. Contributions are also scaled by the inverse PDF of the source rays, to account for their distribution. This operation is biased, but it works well in practice.

The final step in the reflection pipeline is a simple bilateral filter that cleans up some of the remaining noise. While this kind of filter can be a blunt instrument that can overblur the image, it is needed for high-roughness reflections. Compared to SSR, ray tracing cannot rely on a blurred version of the screen for prefiltered radiance. It produces much more noise compared to SSR, therefore more aggressive filters are required. Nonetheless, we can still control the filter's effect. We estimate variance in the image during the spatial reconstruction pass, as shown in Figure 15, and use it to tune the bilateral kernel. Where variance is low, we reduce the kernel size and sample count, which prevents overblurring.

Near the end of the frame, we apply temporal antialiasing and get a pretty clean image. When looking at Figure 9, it is important to remember that it comes from a quarter reflection ray per pixel per frame and works with a dynamic camera and dynamic objects.

Since we rely on stochastic sampling to generate smooth to rough reflections, our approach is inherently noisy. Though stochastic sampling is prone to noise, it produces the correct answer given enough samples. An alternative approach could be to blur mirror-like reflections for high roughness. Such a post-filter could work but may introduce bleeding. Filtering also requires a wide pixel footprint to generate blurry reflections, and it may still produce noisy output from high-frequency details. Structured aliasing is difficult to filter as well, so non-stochastic effects can produce

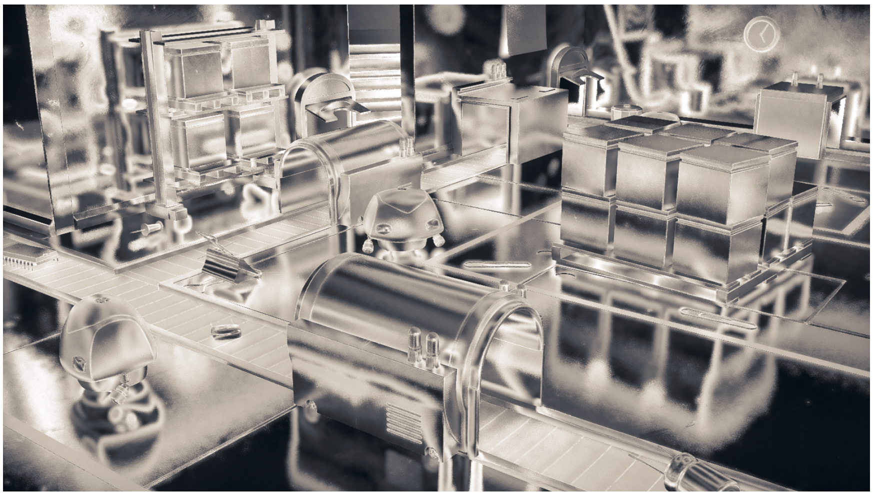


Figure 15: Reflection variance.

more flickering than stochastic ones. In parallel, stochastic techniques can amplify variance in a scene, especially for small bright sources. Tiny bright sources could be detected and handled with more bias, shifting the algorithm toward a non-stochastic approach. Additional research here is required. Our reflection pipeline is already a step in this direction, combining stochastic sampling with spatial reconstruction. In practice, we bias our primary sample space so that rays fly a bit closer to the mirror direction, and we then cancel some of this bias during filtering.

For temporal accumulation a simple exponential smoothing operator, which blends on top of the previous frame, is not sufficient. Movement is particularly difficult for temporal techniques, as reprojection has to correlate results between frames. Two different methods first come to mind when reprojecting reflections. First, we can use the motion vectors of the reflector, which we can inherently reuse from other techniques in the hybrid pipeline. Second, reflections move with their own parallax, can be tracked by finding the average length of the reflection rays, and can be reprojected via an average hit point for each pixel. Both approaches are shown in Figure 16.

Separately each method has its advantages. As shown in Figure 16, motion vectors work well for rough and curved surfaces but fail with shiny flat surfaces. Hit point reprojection, on the other hand, works for the floor but fails on curved surfaces. Alternatively, we can build simple statistics of every pixel in the newly generated image and use that to choose which reprojection approach to take. If we calculate the mean color and standard deviation of every new pixel, a distance metric can be defined and used to weigh the reprojected values:

```
dist = (rgb - rgb_mean) / rgb_deviation;
w = exp2(-10 * luma(dist));
```

Finally, as demonstrated by Karis [15], we can use local pixel statistics to reject or clamp the reprojected values, and force them to fit the new distribution. While the results are not perfect, it is certainly a step forward. This biases the result and can create some flickering, but it nicely cleans up the ghosting and is sufficient for

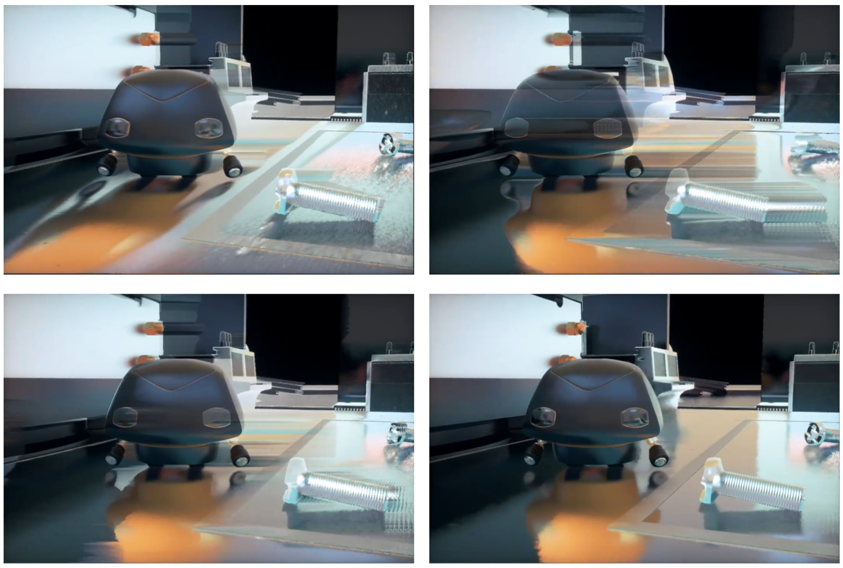


Figure 16: Top left: motion reprojection. Top right: hit point reprojection. Bottom left: motion and hit point reprojection blending. Bottom right: with reprojection clamping.

real-time purposes.

2.3 Ambient Occlusion

In offline and real-time graphics, ambient occlusion (AO) [18] is used to improve near field rendering, where the general global illumination solution fails. This can improve perceived quality and ground objects where little direct shadowing is visible. In video games, AO is often either precalculated offline or computed in real time using screen-space information. Baking can provide accurate results, but fails to account for dynamic geometry. Screen-space techniques such as ground-truth ambient occlusion (GTAO) [14] and horizon-based ambient occlusion (HBAO) [5] can produce convincing results, but are limited by the information available on screen. The failure of screen-space techniques can be quite jarring, especially if offscreen geometry should be affecting occlusion. The same is true if such geometry is inside the view frustum but is occluded.

With real-time ray tracing, we can calculate high-quality ambient occlusion in a way that is free from the constraints of the raster-based techniques just mentioned. In *PICA PICA*, we stochastically sample the occlusion function by generating rays randomly across the hemisphere. To reduce noise, we sample with a cosine-weighted distribution [9]. We also expose the maximum ray distance as a configurable variable per scene, for performance but also visual-quality purposes. To further reduce noise, we filter the raw ray traced ambient occlusion with a technique similar to the one used for our ray traced shadows.

```
// Partial code for AO ray generation shader, truncated for brevity.
// The full shader is otherwise essentially identical to the shadow
// ray generation.
float result = 0;

for (uint i = 0; i < numRays; i++)
{
  // Select a random direction for our AO ray.
  float rnd1 = frac(haltonNext(hState) + randomNext(frameSeed));
  float rnd2 = frac(haltonNext(hState) + randomNext(frameSeed));
  float3 rndDir = cosineSampleHemisphere(rnd1, rnd2);

  // Rotate the hemisphere.
  // Up is in the direction of the pixel surface normal.
  float3 rndWorldDir = mul(rndDir, createBasis(gbuffer.worldNormal));

  // Create a ray and payload.
  ShadowData shadowPayload;
  shadowPayload.miss = false;

  RayDesc ray;
  ray.Origin = position;
  ray.Direction = rndWorldDir;
  ray.TMin = g_aoConst.minRayLength;
  ray.TMax = g_aoConst.maxRayLength;

  // Trace our ray;
  // use the shadow miss, since we only care if we miss or not.
  TraceRay(g_rtScene,
           RAY_FLAG_SKIP_CLOSEST_HIT_SHADER|
               RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH,
           RaytracingInstanceMaskAll,
           HitType_Shadow,
           SbtRecordStride,
           MissType_Shadow,
           ray,
           shadowPayload);

  result += shadowPayload.miss ? 1 : 0;
}

result /= numRays;
```

The shader code for ray traced ambient occlusion is similar to that of shadows, and as such we only list the part specific to AO here. As with shadows, we reconstruct the world-space position and normal for each pixel visible on screen using the G-buffer.

Since the `miss` flag in the shadow payload is initialized to false and is only set to true in the miss shader, we can set `RAY_FLAG_SKIP_CLOSEST_HIT_SHADER` to skip the hit shader, for performance. We also do not care about how far away an intersection is. We just want to know if there is an intersection, so we use `RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH` as well. Finally, the cosine-weighted distribution of samples is generated on a unit hemisphere and rotated into world space using a basis produced from the G-buffer normal.

In Figure 17, a comparison between different versions of ambient occlusion can be seen, with a maximum ray length of 0.6 meters. In the top left ground truth was generated by sampling with 1000 samples per pixel (spp). This is too slow for real time. In *PICA PICA*, we sample with one or two rays per pixel, which produces

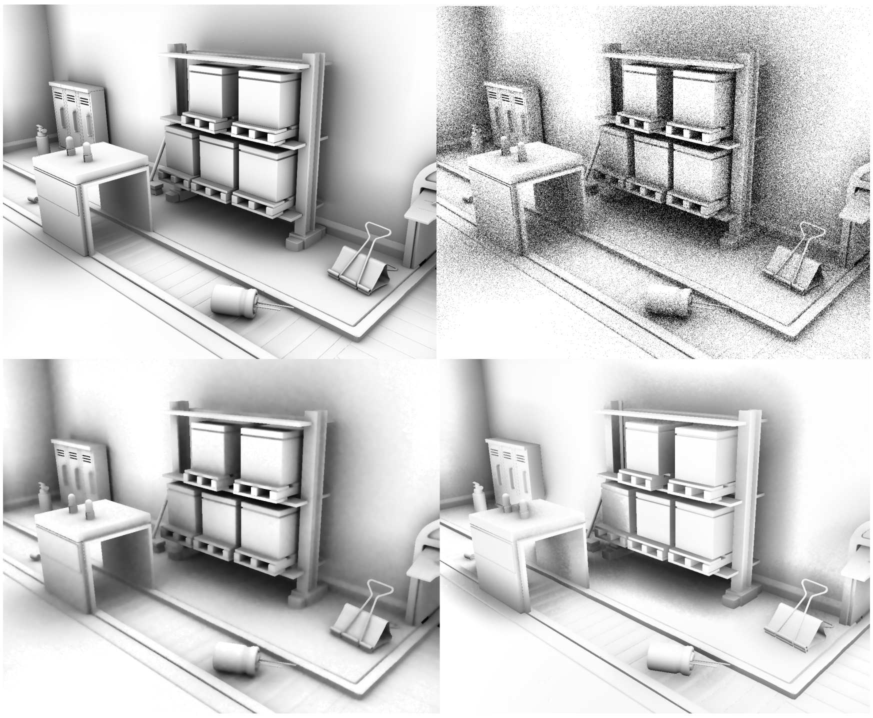


Figure 17: Top left: ray traced AO (1000 spp). Top right: hybrid ray traced AO (1 spp). Bottom left: filtered hybrid ray traced AO (1 spp). Bottom right: GTAO.

the rather noisy result seen in the top right of Figure 17. After applying our filter, the results are more visually pleasing, as seen in the bottom left part of the same figure. Our filtered ray traced ambient occlusion matches the reference well, albeit a bit less sharp, with only one ray per pixel.

2.4 Transparency

Unlike rasterization, where the rendering of transparent geometry is often treated separately from opaque geometry, ray tracing can streamline and unify the computation of light transport inside thick media with the rest of the scene. One notable example is the rendering of realistic refractions for transparent surfaces such as glass. See Figure 18.

With ray tracing, interface transitions are easier to track because each transition is part of a chain of intersections. As seen in Figure 19, as a ray travels inside and then outside a medium, it can be altered based on the laws of optics and parameters of that medium. Intermediary light transport information is modified and carried with the ray, as part of its payload, which enables the computation of visually convincing effects such as absorption and scattering. We describe the latter in Section 2.5.

When tracking medium transitions, ray tracing enables order-independent transparency and exact sorting of transparent meshes with other scene geometry. While order-independent smooth refractions are straightforward, rough refractions are also

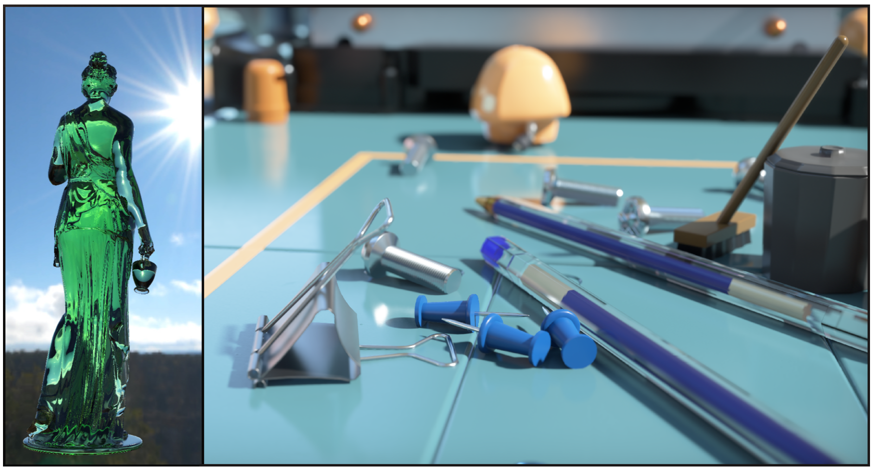


Figure 18: Left: object-space ray traced transparency result. Right: texture-space output.

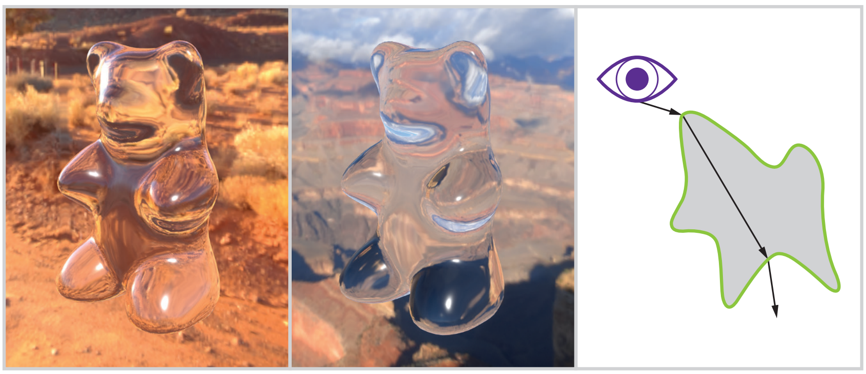


Figure 19: Object-space ray traced smooth transparency.

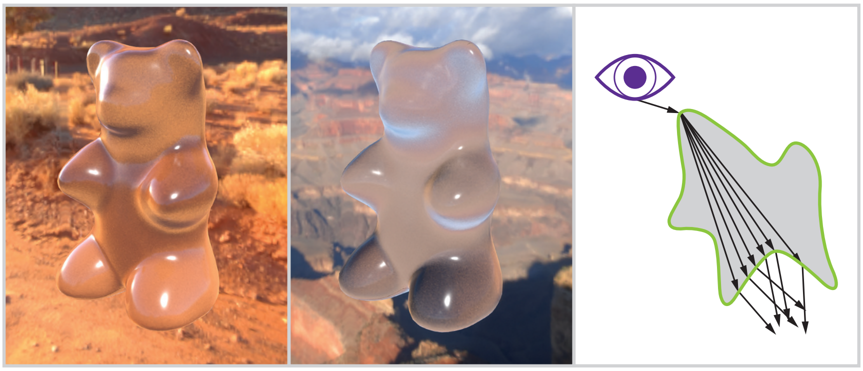


Figure 20: Object-space ray traced rough transparency.

possible but require additional care. As shown in Figure 20, multiple samples are necessary in order to converge rough refractions to a noise-free result. Such refractions are difficult to filter in an order-independent fashion, due to the possibility of multiple layers overlapping on screen. Successful denoisers today assume just one layer of surfaces, making screen-space denoising intractable for order-independent transparency. Additionally, depending on scene complexity, per-pixel order-independent transparency can also be quite memory-intensive and its performance intractable.

To palliate this, we adopt a hybrid approach that combines object-space ray tracing with texture-space parameterization and integration. Textures provide a stable integration domain, as well as a predictable memory footprint. Object-space parameterization in texture space for ray tracing also brings a predictable number of rays per object per frame and can therefore be budgeted. This level of predictability is essential for real-time constraints. An example of this texture-space parameterization, generated on demand prior to ray tracing, is presented in Figure 21. Our approach minimally requires positions and normals, but additional surface and material parameters can be stored in a similar fashion. This is akin to having per-object G-buffers. A non-overlapping UV unwrap is also required. The ray traced result is shown in Figure 22.

Using our parameterization and camera information, we drive ray origin and ray direction during tracing. Clear glass refraction is achieved using Snell's law, whereas rough glass refraction is achieved via a physically based scattering function [28]. The latter generates rays refracted off microfacets, spreading into wider cones for rougher interfaces.

A feature of DXR that enables this technique is the ability to know if we have transitioned from one medium to another. This information is provided by the `HitKind()` function, which informs us if we have hit the front or back side of the geometry:

```
// If we are going from air to glass or glass to air,
// choose the correct index of refraction ratio.
bool isBackFace = (HitKind() == HIT_KIND_TRIANGLE_BACK_FACE);
float ior = isBackFace ? iorGlass / iorAir : iorAir / iorGlass;

RayDesc refractionRay;
```

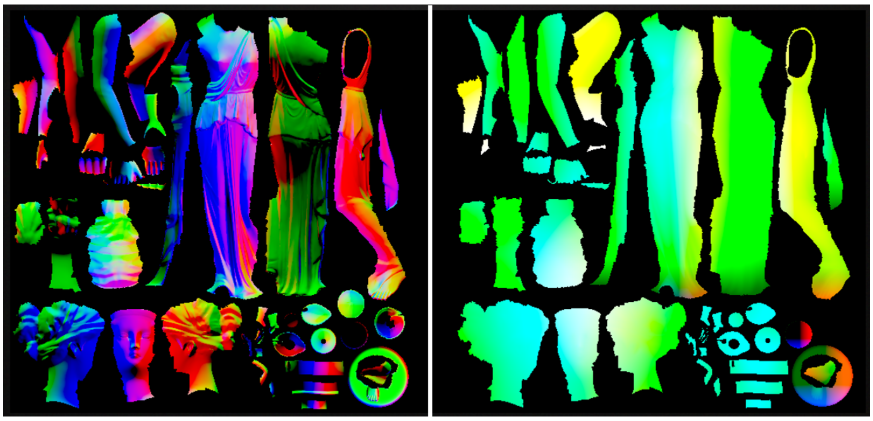


Figure 21: Object-space parameterization: normals (left) and positions (right).

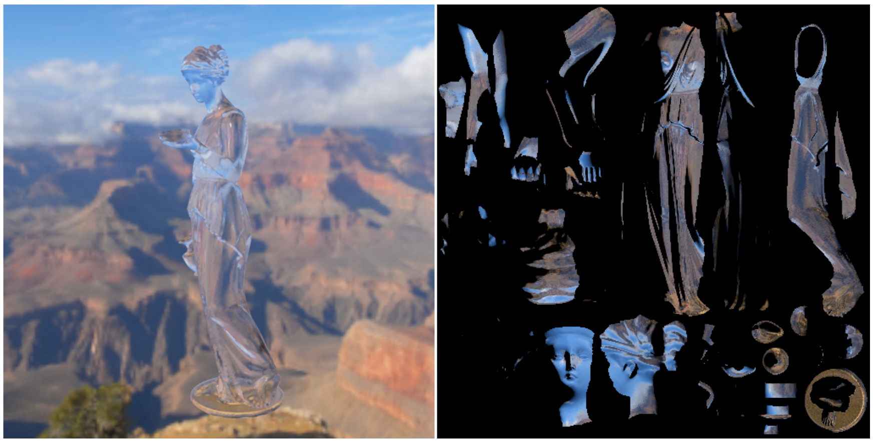


Figure 22: Object-space ray traced transparency: result (left) and texture-space output (right).

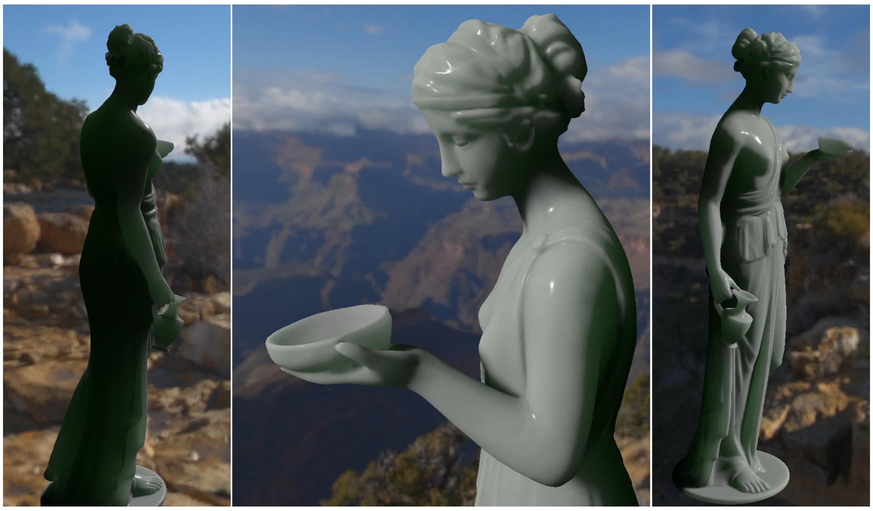


Figure 23: Ray traced translucency.

```
refractionRay.Origin = worldPosition;
refractionRay.Direction = refract(worldRayDir, worldNormal, ior);
```

With such information we can alter the index of refraction and correctly handle media transitions. We can then trace a ray, sample lighting, and finish by modulating the results by the medium's absorption, approximated by Beer's law. Chromatic aberration can also be applied, to approximate wavelength-dependent refraction.

This process is repeated recursively, with a recursion limit set depending on performance targets.

2.5 Translucency

Three ray traced images with translucency are shown in Figure 23. Similar to transparency, we parameterize translucent objects in texture space. The scattering process is represented in Figure 24: Starting with (a) a light source and a surface, we consider valid vectors using (b) the surface normals. Focusing on a single normal vector for now, (c) we then push the vector inside the surface. Next, (d) we launch rays in a uniform sphere distribution similar to the work by Christensen et al. [6]. Several rays can be launched at once, but we only launch one per frame. Finally, (e) lighting is computed at the intersection, and (f) previous results are gathered and blended with the current result.

We let results converge over multiple frames via temporal accumulation. See Figure 25. Spatial filtering can be used as well, although we did not encounter enough noise to make it worthwhile because of the diffuse nature of the effect. Since lighting conditions can change when objects move, the temporal filter needs to invalidate results and adapt to dynamism. A simple exponential moving average here can be sufficient. For improved response and stability, we use an adaptive temporal filter based on exponential averaging [26], which is described further in the next section and which varies its hysteresis to quickly reconverge to dynamically changing conditions.

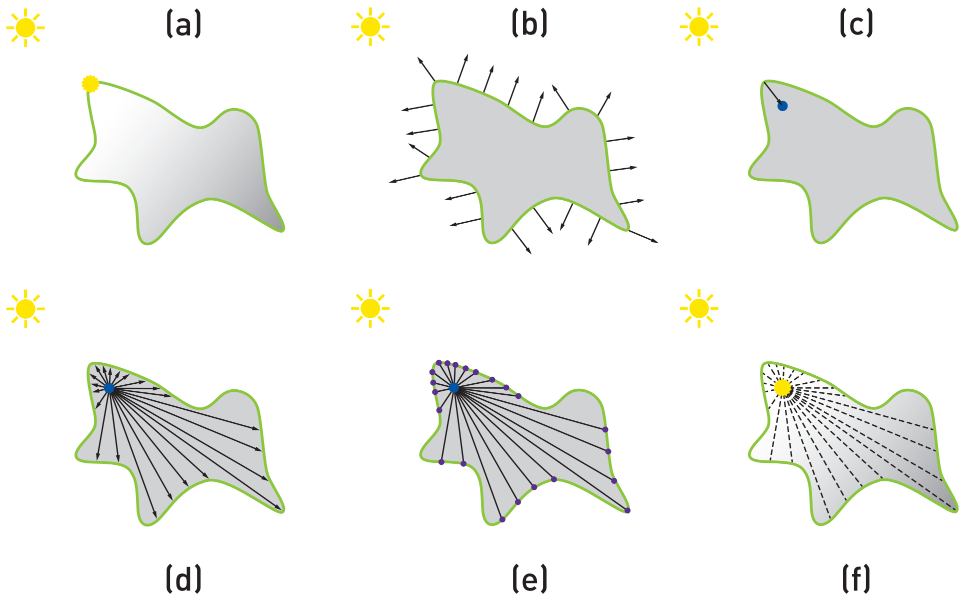


Figure 24: Light scattering process. (See text for details.)

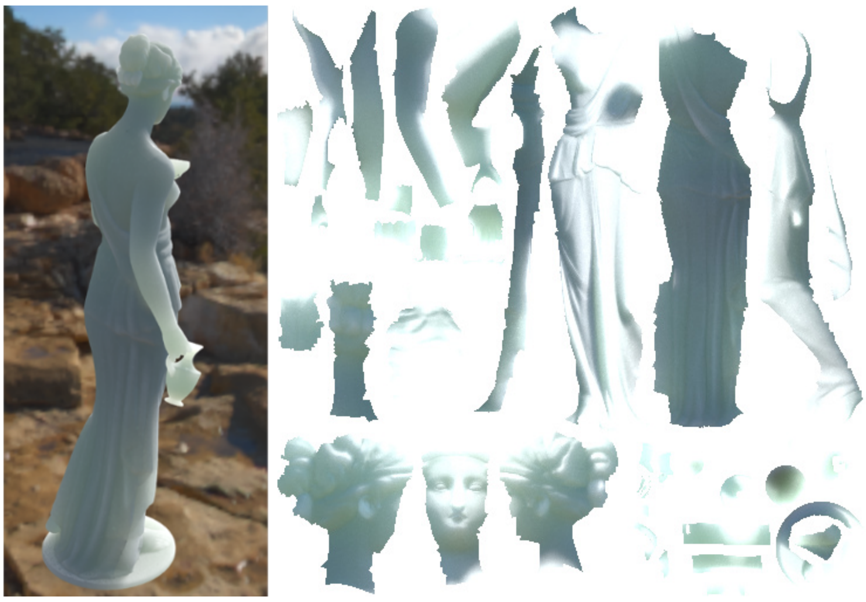


Figure 25: Texture-space ray traced translucency accumulation.

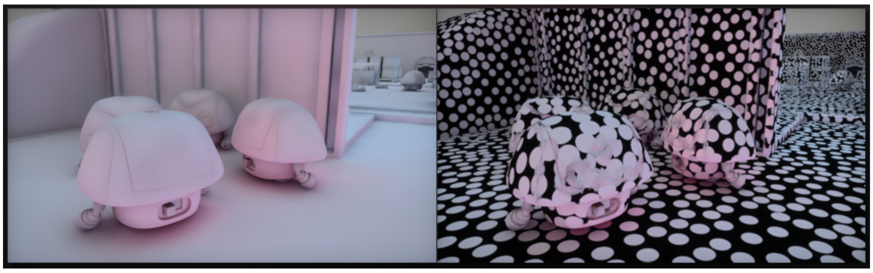


Figure 26: Surfel-based diffuse interreflection.

2.6 Global Illumination

As part of global illumination (GI), indirect lighting applied in a diffuse manner to surfaces makes scene elements fit with each other, and provides results representative of reality.

PICA PICA features an indirect diffuse lighting solution that does not require any precomputation or pre-generated parameterization, such as UV coordinates. This reduces the mental burden on artists, and provides realistic results by default, without them having to worry about implementation details of the GI system.

It supports dynamic and static scenes, is reactive, and refines over time to a high-quality result. Since solving high-quality per-pixel GI every frame is not currently possible for real-time rates, spatial or temporal accumulation is required. For this project, 250,000 rays per frame are budgeted for diffuse interreflections.

To achieve this performance target at quality, a world-space structure of dynamically distributed surfels is created. See Figure 26. For this scene we use up to 250,000 surfels, corresponding to one ray per surfel per frame. Each surfel is represented by a position, normal, radius, and irradiance. Persistent in world space, results accumulate over time without disocclusion issues. As it is a freeform cloud of surfels, no parameterization of the scene is necessary. In the case of animated objects, surfels remember the object on which they were spawned and are updated every frame.

A pre-allocated array of surfels is created at startup. Surfels are then spawned progressively, based on the view camera. See Figure 27. The latter step is done on the GPU, using an atomic counter incremented as surfels get assigned. The surfel placement algorithm uses G-buffer information and is an iterative process. We start by calculating the coverage of each pixel by the current surfel set, in a 16×16 tile. We are interested in pixels with low coverage because we would like to spawn new surfels there. To find the best candidates, the worst coverage is chosen first. We detect it by subdividing the screen into tiles and finding the lowest coverage in each tile. Once found, we can spawn a surfel at the pixel's location using the G-buffer normal and depth. The pixel is then added to the surfel structure.

It is important to note that surfels are spawned probabilistically. In the event where the camera moves close to a wall that is missing surfels, suddenly all pixels have low coverage and will require surfels. This would end up creating a lot of surfels in a small area, since screen tiles are independent of each other. To solve this issue, the spawn heuristic is made proportional to the pixel's projected area in world space. This process runs every frame and continues spawning surfels wherever coverage is

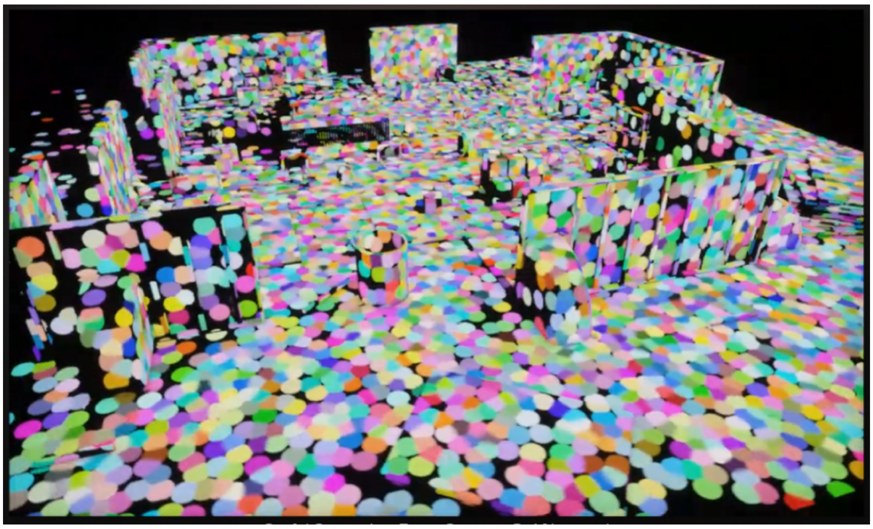


Figure 27: Surfels progressively allocated to the scene.

low. Additionally, since surfels are allocated based on screen-space constraints, sudden geometric or camera transitions to first-seen areas can show missing diffuse interreflections. This "first frame" problem is common among techniques that rely on temporal amortization, and it could be noticed by the user. The latter was not an issue for *PICA PICA*, but it could be depending on the target usage of this approach.

Once assigned, surfels are persistent in the array and scene. See Figure 28. This is necessary for the incremental aspect of the diffuse interreflection accumulation. Because of the simple nature of *PICA PICA*'s scene, we did not have to manage complex surfel recycling. We simply reset the atomic counter at scene reload. As shown in Section 3, performance on current ray tracing hardware was quite manageable, at a cost of 0.35 ms for 250,000 surfels. We believe surfel counts can be increased quite a bit before it becomes a performance issue. A more advanced allocation and deallocation scheme might be necessary in case one wants to use this technique for a more complex use case, such as a video game. Further research here is required, especially with regards to level of detail management for massive open-world games.

Surfels are rendered similarly to light sources when applied to the screen. Similar to the approach by Lehtinen et al. [19], a smoothstep distance attenuation function is used, along with the Mahalanobis metric to squash surfels in the normal direction. Angular falloff is used as well, but each surfel's payload is just irradiance, without any directionality. For performance reasons, an additional world-space data structure enables the query of indirect diffuse illumination in three-dimensional space. This grid structure, in which every cell stores a list of surfels, also serves as a culling mechanism. Each pixel or point in space can then look up the containing cell and find all relevant surfels.

A downside of using surfels is, of course, the limited resolution and the lack of high-frequency detail. To compensate, a colored multiple-bounce variant of screen-

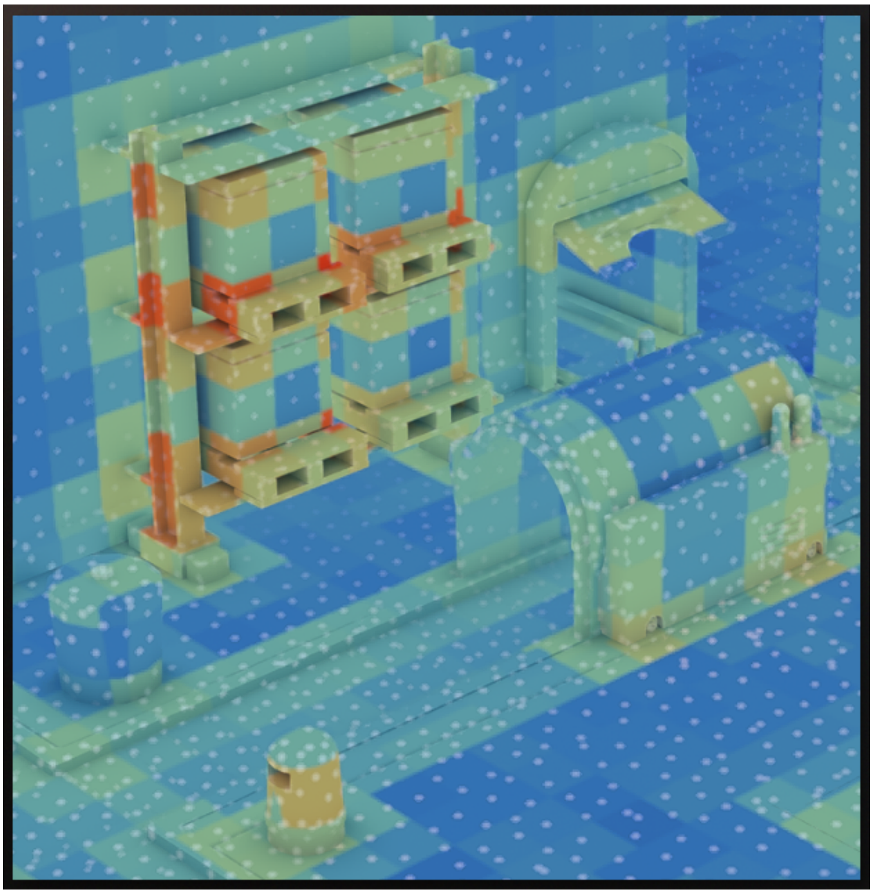


Figure 28: Surfel screen application.

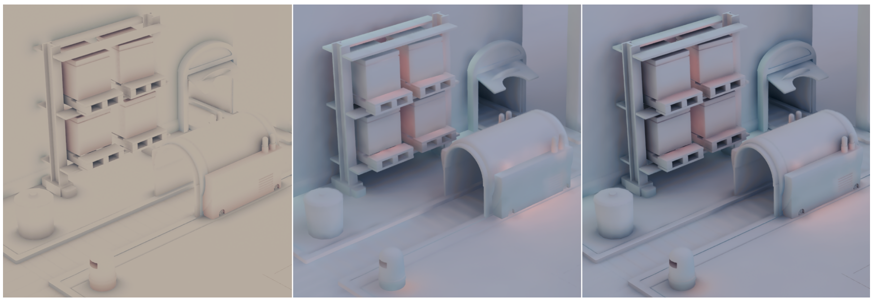


Figure 29: Left: colored GTAO. Center: surfel GI. Right: surfel GI with colored GTAO.

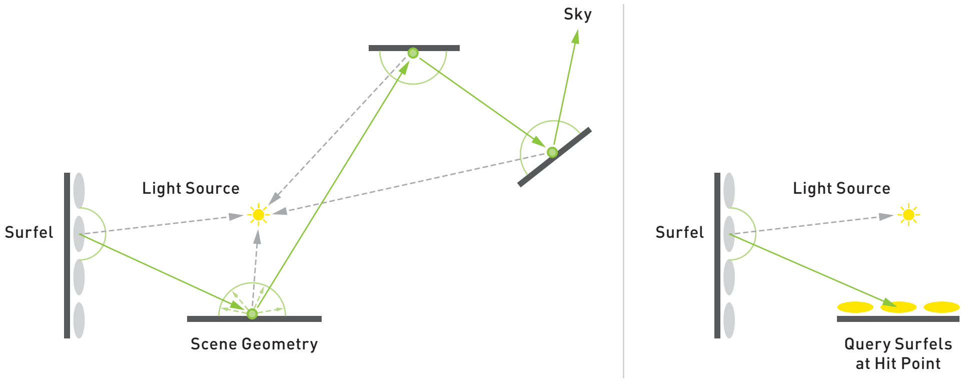


Figure 30: Left: full recursive path tracing. Right: incremental previous frame path tracing.

space ambient occlusion [14] is applied to the calculated per-pixel irradiance. The use of high-frequency AO here makes our technique diverge from theory, but it is an aesthetic choice that compensates for the lack of high-frequency detail. This colored multi-frequency approach also helps retain the warmth in our toy-like scenes. See Figure 29.

Surfel irradiance is calculated by building a basic unidirectional path tracer with explicit light connections. More paths are allocated to newly spawned surfels, so that they converge quickly, and then slowly the sample rate is decreased to one path per frame. Full recursive path tracing is a bit expensive, and for our use case quite unnecessary. We can exploit temporal coherence by reusing previous outputs and can amortize the extra bounces over time. We limit path length to just one edge by shooting a single ray and immediately sampling the previous frame's results, as shown in Figure 30. The surfels path trace one bounce with indirect shading coming from other surfels at that bounce (converging over time), instead of going for a full multiple-bounce path. Our approach is much closer to radiosity than path tracing, but the visual results are similar in our mostly-diffuse scenes.

Path tracing typically uses Monte Carlo integration. If expressed as a running mean estimator, the integration is an average of contributions with linearly decaying weights. Its convergence hinges on the integrand being immutable. In the case of our dynamic GI, the integrand changes all the time. Interactive path tracers and progressive light map bakers [8, 13] typically tackle this by resetting accumulation on change. Their goals are different though, as they try to converge to the correct solution over time, and do not allow error. As such, a hard reset is actually desirable for them, but not for a real-time demo.

Since we cannot use proper Monte Carlo, we outright give up on ever converging. Instead, we use a modified exponential mean estimator,

$$\begin{aligned} \overline{x}_0 &= 0, \\ \overline{x}_{n+1} &= \texttt{lerp}(\overline{x}_n, x_{n+1}, k), \end{aligned} \tag{1}$$

whose formulation is similar to that of plain Monte Carlo. The difference is in how the blending factor k is defined. In exponential averaging, the weight for a new sample is constant and typically set low, so that variance in input is scaled by a small value and does not look jarring in the output.

If the input does not have high variance, the output will not either. We can then use a higher blending factor k. The specifics of our integrand change all the time though, so we need to estimate that dynamically. We run short-term mean and variance estimators, which we then use to inform our primary blending factor. The short-term statistics also give us an idea of the plausible range of values into which the inputs samples should fall. When they start to drift, we increase the blending factor. This works well in practice and allows for a reactive indirect diffuse lighting solution, as demonstrated by this demo.

```
struct MultiscaleMeanEstimatorData
{
  float3 mean;
  float3 shortMean;
  float vbbr;
  float3 variance;
  float inconsistency;
};

float3 MultiscaleMeanEstimator(float3 y,
  inout MultiscaleMeanEstimatorData data,
  float shortWindowBlend = 0.08f)
{
  float3 mean = data.mean;
  float3 shortMean = data.shortMean;
  float vbbr = data.vbbr;
  float3 variance = data.variance;
  float inconsistency = data.inconsistency;

  // Suppress fireflies.
  {
    float3 dev = sqrt(max(1e-5, variance));
    float3 highThreshold = 0.1 + shortMean + dev * 8;
    float3 overflow = max(0, y - highThreshold);
    y -= overflow;
  }

  float3 delta = y - shortMean;
  shortMean = lerp(shortMean, y, shortWindowBlend);
  float3 delta2 = y - shortMean;

  // This should be a longer window than shortWindowBlend to avoid bias
  // from the variance getting smaller when the short-term mean does.
```

```
    float varianceBlend = shortWindowBlend * 0.5;
    variance = lerp(variance, delta * delta2, varianceBlend);
    float3 dev = sqrt(max(1e-5, variance));

    float3 shortDiff = mean - shortMean;

    float relativeDiff = dot( float3(0.299, 0.587, 0.114),
          abs(shortDiff) / max(1e-5, dev) );
    inconsistency = lerp(inconsistency, relativeDiff, 0.08);

    float varianceBasedBlendReduction =
          clamp( dot( float3(0.299, 0.587, 0.114),
          0.5 * shortMean / max(1e-5, dev) ), 1.0/32, 1 );

    float3 catchUpBlend = clamp(smoothstep(0, 1,
          relativeDiff * max(0.02, inconsistency - 0.2)), 1.0/256, 1);
    catchUpBlend *= vbbr;

    vbbr = lerp(vbbr, varianceBasedBlendReduction, 0.1);
    mean = lerp(mean, y, saturate(catchUpBlend));

    // Output
    data.mean = mean;
    data.shortMean = shortMean;
    data.vbbr = vbbr;
    data.variance = variance;
    data.inconsistency = inconsistency;

    return mean;
}
```

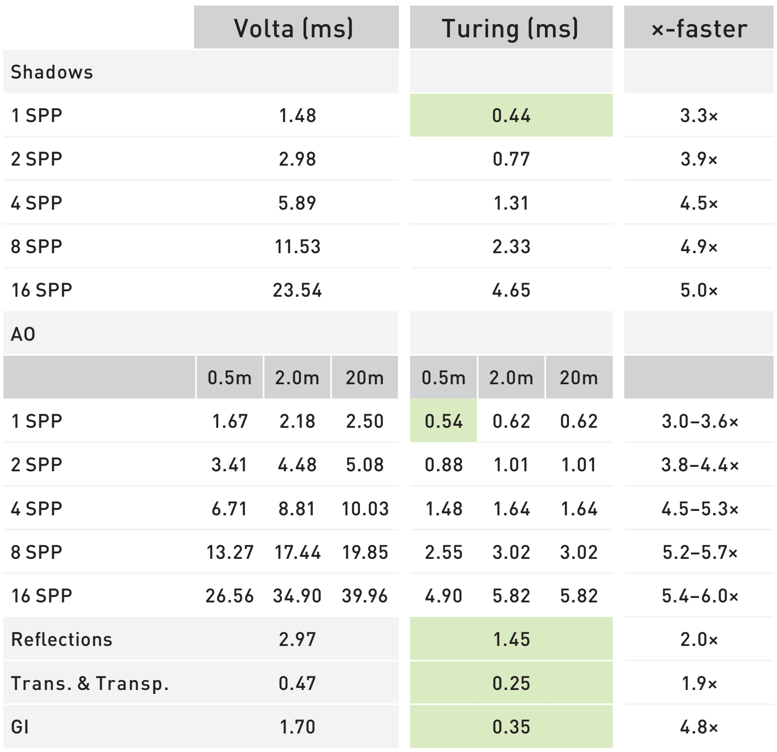
3 Performance

Here we provide various performance numbers behind the ray tracing aspect of our hybrid rendering pipeline. The numbers in Figure 31 were measured on pre-release NVIDIA Turing hardware and drivers, for the scene and view shown in Figure 32. When presented at SIGGRAPH 2018 [4], *PICA PICA* ran at 60 frames per second (FPS), at a resolution of 1920 × 1080. Performance numbers were also captured against the highest-end GPU at that time, the NVIDIA Titan V (Volta).

4 Future

The techniques in *PICA PICA*'s hybrid rendering pipeline enable real-time visually pleasing results with (almost) path traced quality, while being mostly free from noise in spite of relatively few rays being traced per pixel and per frame. Real-time ray tracing makes it possible to replace finicky hacks with unified approaches, allowing for the phasing-out of artifact-prone algorithms such as screen-space ray marching, along with all the artist time required to tune them. This opens the door to truly effortless photorealism, where content creators do not need to be experts in order to get high-quality results.

The surface has been barely scratched, and with real-time ray tracing a new world of possibilities opens up. While developers will always keep asking for more power, the hardware that we have today already allows for high-quality results at



	Volta (ms)			Turing (ms)			x-faster
Shadows							
1 SPP	1.48			0.44			3.3×
2 SPP	2.98			0.77			3.9×
4 SPP	5.89			1.31			4.5×
8 SPP	11.53			2.33			4.9×
16 SPP	23.54			4.65			5.0×
AO							
	0.5m	2.0m	20m	0.5m	2.0m	20m	
1 SPP	1.67	2.18	2.50	0.54	0.62	0.62	3.0–3.6×
2 SPP	3.41	4.48	5.08	0.88	1.01	1.01	3.8–4.4×
4 SPP	6.71	8.81	10.03	1.48	1.64	1.64	4.5–5.3×
8 SPP	13.27	17.44	19.85	2.55	3.02	3.02	5.2–5.7×
16 SPP	26.56	34.90	39.96	4.90	5.82	5.82	5.4–6.0×
Reflections	2.97			1.45			2.0×
Trans. & Transp.	0.47			0.25			1.9×
GI	1.70			0.35			4.8×

Figure 31: Performance measurements in milliseconds (ms). SIGGRAPH 2018 timings are highlighted in green.

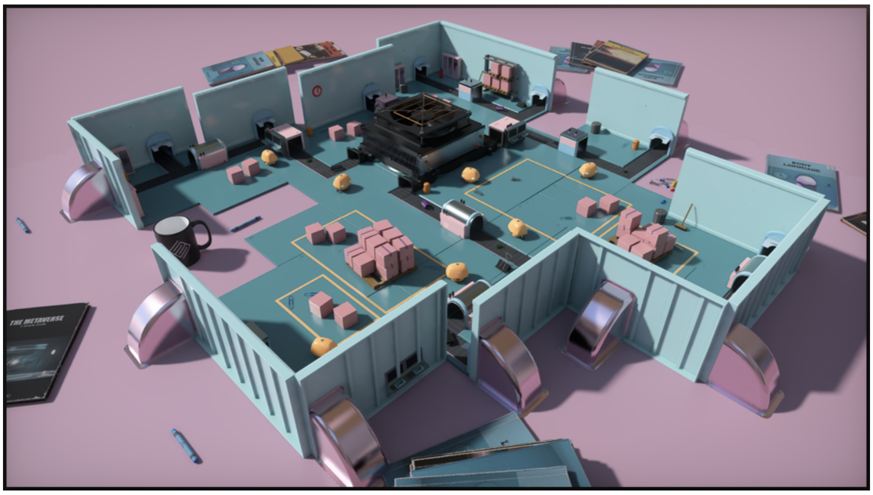


Figure 32: Performance scene.

real-time performance rates. If ray budgets are devised wisely, with hybrid rendering we can approach the quality of offline path tracers in real time.

5 Code

```
struct HaltonState
{
  uint dimension;
  uint sequenceIndex;
};

void haltonInit(inout HaltonState hState,
                int x, int y,
                int path, int numPaths,
                int frameId,
                int loop)
{
  hState.dimension = 2;
  hState.sequenceIndex = haltonIndex(x, y,
        (frameId * numpaths + path) % (loop * numpaths));
}

float haltonSample(uint dimension, uint index)
{
  int base = 0;

  // Use a prime number.
  switch (dimension)
  {
  case 0:  base = 2;   break;
  case 1:  base = 3;   break;
  case 2:  base = 5;   break;
  [...] // Fill with ordered primes, case 0-31.
  case 31: base = 131; break;
  default: base = 2;   break;
  }

  // Compute the radical inverse.
  float a = 0;
  float invBase = 1.0f / float(base);

  for (float mult = invBase;
       sampleIndex != 0; sampleIndex /= base, mult *= invBase)
  {
    a += float(sampleIndex % base) * mult;
  }

  return a;
}

float haltonNext(inout HaltonState state)
{
  return haltonSample(state.dimension++, state.sequenceIndex);
}

// Modified from [pbrt]
uint haltonIndex(uint x, uint y, uint i)
{
  return ((halton2Inverse(x % 256, 8) * 76545 +
      halton3Inverse(y % 256, 6) * 110080) % m_increment) + i * 186624;
}

// Modified from [pbrt]
uint halton2Inverse(uint index, uint digits)
{
  index = (index << 16) | (index >> 16);
  index = ((index & 0x00ff00ff) << 8) | ((index & 0xff00ff00) >> 8);
  index = ((index & 0x0f0f0f0f) << 4) | ((index & 0xf0f0f0f0) >> 4);
  index = ((index & 0x33333333) << 2) | ((index & 0xcccccccc) >> 2);
  index = ((index & 0x55555555) << 1) | ((index & 0xaaaaaaaa) >> 1);
  return index >> (32 - digits);
}

// Modified from [pbrt]
uint halton3Inverse(uint index, uint digits)
{
  uint result = 0;
  for (uint d = 0; d < digits; ++d)
  {
    result = result * 3 + index % 3;
    index /= 3;
  }
  return result;
}
```

Acknowledgments

The authors would like to thank the *PICA PICA* team at SEED, a technical and creative research division of Electronic Arts. We would also like to thank our friends at Frostbite and DICE, with whom we have had great discussions and collaboration as we built this hybrid rendering pipeline. Moreover, this endeavor would not have been possible without support from the folks at NVIDIA and the DirectX team at Microsoft. The authors would also like to thank Morgan McGuire for his review of this chapter, as well as Tomas Akenine-Möller and Eric Haines for their insight and support.

References

[1] Akenine-Möller, T., Haines, E., Hoffman, N., Pesce, A., Iwanicki, M., and Hillaire, S. *Real-Time Rendering*, fourth ed. A K Peters/CRC Press, 2018.

[2] Andersson, J., and Barré-Brisebois, C. DirectX: Evolving Microsoft's Graphics Platform. Microsoft Sponsored Session, Game Developers Conference, 2018.

[3] Andersson, J., and Barré-Brisebois, C. Shiny Pixels and Beyond: Real-Time Raytracing at SEED. NVIDIA Sponsored Session, Game Developers Conference, 2018.

[4] Barré-Brisebois, C., and Halén, H. PICA PICA and NVIDIA Turing. NVIDIA Sponsored Session, SIGGRAPH, 2018.

[5] Bavoil, L., Sainz, M., and Dimitrov, R. Image-Space Horizon-Based Ambient Occlusion. In *ACM SIGGRAPH Talks* (2008), p. 22:1.

[6] Christensen, P., Harker, G., Shade, J., Schubert, B., and Batali, D. Multiresolution Radiosity Caching for Global Illumination in Movies. In *ACM SIGGRAPH Talks* (2012), p. 47:1.

[7] Cranley, R., and Patterson, T. Randomization of Number Theoretic Methods for Multiple Integration. *SIAM Journal on Numerical Analysis 13*, 6 (1976), 904–914.

[8] Dean, M., and Nordwall, J. Make It Shiny: Unity's Progressive Lightmapper and Shader Graph. Game Developers Conference, 2016.

[9] Dutré, P., Bekaert, P., and Bala, K. *Advanced Global Illumination.* A K Peters, 2006.

[10] EA SEED. Project PICA PICA—Real-Time Raytracing Experiment Using DXR (DirectX Raytracing). https://www.youtube.com/watch?v=LXo0WdlELJk, March 2018.

[11] Fong, J., Wrenninge, M., Kulla, C., and Habel, R. Production Volume Rendering. Production Volume Rendering, SIGGRAPH Courses, 2017.

[12] 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.

[13] Hillaire, S. Real-Time Raytracing for Interactive Global Illumination Workflows in Frostbite. NVIDIA Sponsored Session, Game Developers Conference, 2018.

[14] Jiménez, J., Wu, X., Pesce, A., and Jarabo, A. Practical Real-Time Strategies for Accurate Indirect Occlusion. Physically Based Shading in Theory and Practice, SIGGRAPH Courses, 2016.

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

[16] Lagarde, S. Memo on Fresnel Equations. Blog, April 29, 2013.

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

[18] Landis, H. Production-Ready Global Illumination. RenderMan in Production, SIGGRAPH Courses, 2002.

[19] Lehtinen, J., Zwicker, M., Turquin, E., Kontkanen, J., Durand, F., Sillion, F., and Aila, T. A Meshless Hierarchical Representation for Light Transport. *ACM Transactions in Graphics 27*, 3 (2008), 37:1–37:10.

[20] McGuire, M., and Mara, M. Phenomenological Transparency. *IEEE Transactions on Visualization and Computer Graphics 23*, 5 (2017), 1465–1478.

[21] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*, third ed. Morgan Kaufmann, 2016.

[22] Salvi, M. An Excursion in Temporal Super Sampling. From the Lab Bench: Real-Time Rendering Advances from NVIDIA Research, SIGGRAPH Courses, 2016.

[23] Sandy, M. Announcing Microsoft DirectX Raytracing! DirectX Developer Blog, March 19, 2018.

[24] 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.

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

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

[27] Stachowiak, T., and Uludag, Y. Stochastic Screen-Space Reflections. Advances in Real-Time Rendering, SIGGRAPH Courses, 2015.

[28] Walter, B., Marschner, S. R., Li, H., and Torrance, K. E. Microfacet Models for Refraction through Rough Surfaces. In *Eurographics Symposium on Rendering* (2007), pp. 195–206.

[29] Weidlich, A., and Wilkie, A. Arbitrary Layered Micro-Facet Surfaces. In *GRAPHITE* (2007), pp. 171–178.