CHAPTER 46

RAY TRACING IN CONTROL

Juha Sjöholm,[1] Paula Jukarainen,[1] and Tatu Aalto[2]
[1]NVIDIA
[2]Remedy Entertainment

ABSTRACT

In this chapter, we describe how ray tracing was used in *Control*. We explain how all ray tracing–based effects, including opaque and transparent reflections, near field indirect diffuse illumination, and contact shadows, were implemented. Furthermore, we describe the denoisers tailored for these effects. These effects resulted in exceptional visual quality in the game, while maintaining real-time frame rates.

46.1 INTRODUCTION

Control, launched in 2019, was one of the very first games with ray tracing. Here, ray tracing was utilized in multiple ways to achieve higher visual quality. *Control* uses a hybrid rendering approach, combining rasterization and ray

Figure 46-1. *Control uses ray traced effects, such as reflections and near field indirect diffuse illumination, to add to its unique artistic style. (Image courtesy of DeadEndThrills.)*

A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8

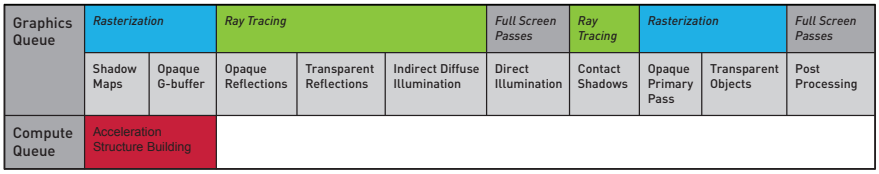

Figure 46-2. *A simplified breakdown of a frame in Control with ray tracing effects enabled showing how both rasterization and ray tracing are used for different purposes.*

tracing. Ray tracing is used in opaque and transparent reflections, near field indirect diffuse illumination, and contact shadows. These effects combined demonstrate that ray tracing can achieve a new level of realism in real-time gaming. A simplified breakdown of a frame can be seen in Figure 46-2.

This introduction section explains common features of the game engine utilized by the different ray tracing effects. The rest of the chapter goes into the details of how each effect, including denoising, was implemented in *Control*. We describe how each effect was optimized to maintain real-time frame rates. We highlight two recurring strategies that were the keys to meeting the performance and quality targets:

- Reducing incoherence by shortening rays when possible.
- Shading incoherent rays at the right level of accuracy, which both reduces noise and improves performance.

46.1.1 NORTHLIGHT ENGINE

Control was developed using the Northlight engine, an in-house game engine developed by Remedy Entertainment. The Northlight engine uses deferred lighting with a bindless material system, which simplifies the implementation and optimization of ray tracing effects, e.g., effects that trace and shade secondary rays (discussed in Sections 46.2, 46.3, and 46.4). It also allows tracing shadow rays for selected light sources for each pixel (discussed in Section 46.5). Additionally, the engine supports an approximative unified parameterization and shading model for all materials, which is used in shading of the ray hits. That works especially well with incoherent rays, as described in Section 46.2.2.

46.1.2 PRECOMPUTED GLOBAL ILLUMINATION

The Northlight engine supports precomputed voxel-based global illumination (see Aalto [1] for details), which had a key role in optimizing the ray traced

reflections and indirect diffuse illumination (as described in Section 46.2.3 and 46.4). The precomputation is performed by a path tracer and is based on static objects and selected light sources. The game levels have the global illumination (GI) data stored in sparse volume textures. The resolution of the available data at a given location is fundamentally artist authored. The data can be sampled with a world-space position and a direction vector. The sampling result is the irradiance over the hemisphere facing the given direction in the given position.

46.1.3 ACCELERATION DATA STRUCTURE BUILDING

All ray tracing passes in the game use the same acceleration data structure. An important principle in the construction of the acceleration data structure is to use the same geometry levels of detail (LODs) as are used in rasterization. This helps with avoiding self-intersections while providing as much detail as possible for ray tracing. A design goal for the ray tracing effects was to provide more accurate details than what is possible with screen space–based techniques executed after rasterizing the scene.

For selecting the objects to be included in the ray tracing acceleration structure, an expanded camera frustum–based culling is applied to the scene objects in order to gather the objects that potentially contribute to some effect. All opaque and most alpha tested objects are included. Some alpha tested vegetation assets are left out as including them would give only minor visual benefits compared to the increased ray tracing costs they incur. Particles that are rendered as opaque meshes are also included in the acceleration data structure. Blended objects and particles are excluded, but that doesn't lead to significant visual issues. However, to support discovering transparent surfaces for rendering reflections on them, blended objects are inserted into the structure with a special cull mask. This is discussed in more detail in Section 46.3.1.

To reduce the memory and cache traffic during ray tracing, the compaction operation available in the DirectX Raytracing (DXR) API is performed on all static acceleration data structures. Skinned meshes are represented as triangle geometries by outputting the skinned vertices to buffers with a compute shader pass. The acceleration structure rebuilds and updates are modulated to achieve better overall performance, but the rasterized and ray traced meshes match on every frame. All acceleration structure building work is executed on the asynchronous compute queue, as shown in Figure 46-2.

46.1.4 LIGHT CLUSTERING

To shade specular reflections or indirect diffuse hits, we require knowledge about which lights affect a given hit location. Evaluating illumination from all scene lights would not be possible simply because of the evaluation cost. The game levels may contain several thousand dynamic lights. For rasterization, effective screen tile–based light culling implementation already existed, but that was not directly suitable for shading ray hits. For ray tracing effects, an additional light clustering pass is executed. It culls the scene lights against cells of an axis-aligned 3D grid in view space. The grid has a limited size that matches the range of the ray tracing effects. The clustering pass stores indices of the lights affecting each grid cell to a texture. When shading the ray hits, the list of lights affecting the grid cell matching the hit position is processed.

46.2 REFLECTIONS

The implementation of ray traced specular reflections in *Control* is straightforward. The reflection rays are generated for each pixel based on the view direction and the surface properties stored in the rasterized G-buffer. Self-intersections are avoided by matching the geometry LODs in ray tracing and rasterization. Additionally, due to the inaccuracy of reconstructing world-space position from the rasterized depth buffer, a bias value scaled by pixel depth toward the camera position and along the surface normal is added to the ray origin. One ray is traced for each pixel excluding only the sky pixels. The game uses the GGX specular bidirectional reflectance distribution function (BRDF) and has surfaces with spatially varying roughness levels. An important design goal was to make the reflections work consistently across all game content. Figure 46-3 shows the reflections on different surfaces.

The following sections describe the techniques that are used to find a good balance between the desired visual quality and performance. Setting the ray length, the fallback solution for missed rays, and shading quality of hits proved to be essential issues. Figure 46-4 illustrates the general workflow.

46.2.1 TRACING REFLECTION RAYS WITH VARYING RAY LENGTH

The ray direction for each pixel is importance-sampled from the GGX distribution. Higher surface roughness means that the ray directions on neighboring pixels are more incoherent, which leads to more noise in the rendered image and higher computation cost. To avoid generating noise that

Figure 46-3. *Left: screen-space reflections. Right: ray traced reflections, which show more accurate details. The denoising process explained in Section 46.6 is applied to the image.*

Figure 46-4. *A general overview of the ray traced reflections rendering.*

would need to be removed from the final image anyway in the denoising passes and also to reduce the cost of the evaluation, the length of the reflection ray is limited based on the surface roughness, as illustrated in Figure 46-5. On surfaces with the maximum roughness value 1.0, the ray length is only about 3 meters. It increases exponentially to about 200 meters as roughness decreases to the minimum value 0.0. These limits were chosen by experimentation to make the result visually plausible. To make reflected objects appear smoothly into the image as they move closer to the reflecting surface, a pixel index–based random variation is also applied to the ray length. This hides visual artifacts from the switch from ray misses to ray hits.

46.2.2 UNIFIED HIT SHADING

For shading the G-buffer, *Control* has a number of material variations with special shading models for, e.g., character skin, eyes, and hair. However, for shading the ray hits in ray tracing effects, a unified variant based on simple parameterization of physically based shading is used for all materials, as

Figure 46-5. *How the reflection ray length varies based on surface roughness. Left: longer reflection rays are generated when the roughness is low and ray direction distribution is coherent. Right: shorter rays are used when the roughness is high and the direction distribution is incoherent.*

Listing 46-1. *Pseudocode overview of the single any-hit shader used to perform alpha testing.*

```
uint material = GetHitMaterialID();
uint3 vertexIndices = GetHitVertexIndices();
float2 uv = InterpolateHitUV(barycentrics, vertexIndices);
float alpha = SampleMaterialAlpha(material, uv);

if (alpha < 0.5f)
    IgnoreHit();
```

mentioned in Section 46.1.1; i.e., the single parameterization and shading model is used for all materials. This allows *Control* to use only one any-hit shader and only one closest-hit shader in the DXR API. Overview of the unified any-hit shader can be seen in Listing 46-1. The unified path is only an approximation, but it provides a visually plausible result in practice. It makes ray tracing development easier in general and helps to achieve satisfying performance especially with incoherent rays and alpha testing. With the incoherent rays, it also reduces noise in the output. The result of the hit shading is the radiance arriving at the G-buffer surface. This is denoised, as described in Section 46.6, before applying it to the receiving surface.

46.2.3 PRECOMPUTED GLOBAL ILLUMINATION FOR RAY MISSES

When the specular reflection rays miss, the precomputed GI data (Section 46.4) is used to approximate the radiance coming from the direction of the ray. The data is sampled at the end of the missed ray. As illustrated in Figure 46-6, the irradiance over the hemisphere provided as the sampling result is converted to average radiance before using it as an approximation. Obviously, this is not accurate for multiple reasons. The result is based only on static geometry and lights, the data resolution is limited, and converting

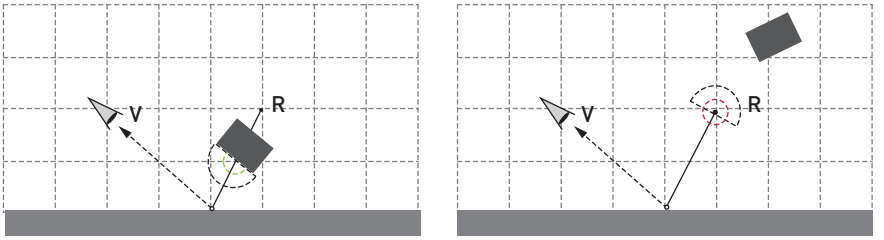

Figure 46-6. *How the precomputed GI data is used for missed rays. Left: the reflection ray hits an object and the hit is shaded. Right: the reflection ray misses the object that is too far away, and the GI data is sampled at the end of the ray as an approximation for the incoming radiance. Sampling the GI data gives irradiance over hemisphere. That is converted into average radiance.*

irradiance to average radiance can cause light leaks. But despite the limitations, this method provides a visually plausible result in practice.

46.2.4 UNIFIED GLOBAL ILLUMINATION SAMPLING FOR HITS AND MISSES

To further unify the shading process, the handling of hits and misses uses partially the same code path. This is possible as the GI data is used to approximate the second ray bounce for hits and the first bounce for misses. For hits, the GI data is sampled directly at the hit location for the second bounce approximation. This means that the hit and miss handling are not done inside hit or miss shaders. In *Control*, the shading happens in a separate compute pass dispatched after the actual ray tracing pass is completed. The separate pass reduces cache pressure especially when the rays are incoherent. It's implemented by resolving the hit geometry normal and texture coordinates in the hit shader and storing them to textures for the shading pass in addition to the hit position and material identifier. The hit position is stored in view space to make half-precision floating-point values sufficient for holding it. The ray misses are marked with a special material identifier. An overview of the compute shader with the unified GI sampling for hits and misses can be seen in Listing 46-2.

46.3 TRANSPARENT REFLECTIONS

Many levels in the game contain a fair amount glass windows, interior walls, and other items with glass surfaces, e.g., wall clocks or poster frames. Having ray traced reflections on those felt like a good addition to the reflections on opaque surfaces. The reflections in different situations are

Listing 46-2. *Pseudocode overview of the shading pass performing the GI sampling for both hits and misses.*

```
float originDepth = DecodeGBufferDepth();
float3 position = DecodeHitOrMissPosition();
uint material = DecodeHitMaterialID();
float3 normal = DecodeHitNormal();
float2 uv = DecodeHitUV();
bool isHit = isMaterialHit(material);

float3 rayDirection = ReconstructRayDirection(originDepth, position);
float3 irradiance = SampleGI(isHit, position, normal, rayDirection);
float3 radiance;

if (isHit) {
    radiance = ShadeHit(position, normal, rayDirection, material, uv,
        irradiance);
}
else {
    radiance = ConvertToAverageRadiance(irradiance);
}

WriteOutput(radiance);
```

shown in Figure 46-7. The following sections describe how the transparent surfaces that receive ray traced reflections are identified, how the reflection rays are generated, and how the results are applied to the receiving surface. Figure 46-8 illustrates the general workflow.

Figure 46-7. *Left: environment map–based reflections on transparent surfaces. Right: ray traced reflections, which show significantly more accurate details.*

Figure 46-8. *A general overview of the ray traced transparent reflections rendering.*

46.3.1 DISCOVERING TRANSPARENT SURFACES

Control uses ray tracing to find transparent surfaces. Primary rays are traced against only the transparent objects. The length of the rays is limited based on the rasterized opaque depth buffer to discover only the visible surfaces, as illustrated in Figure 46-9. If a primary ray hits a transparent surface, a reflection ray is generated based on the surface properties. Otherwise, the pixel is marked as not having a transparent surface.

An alternative to the primary rays approach could have been rasterizing a transparent G-buffer and tracing secondary rays based on that. However, performance of the ray tracing approach proved to be competitive. The directions of the primary rays are naturally coherent, and the processing applied to them is uniform and quite simple in this case.

Transparent objects are inserted into the same acceleration structure as everything else but marked with a different cull mask. Storing them in a separate structure was also tried, but because there isn't a significant overall performance difference between the two approaches in this case, using a common acceleration structure was chosen for simplicity. The overall performance is a combination of the acceleration structure build cost and the ray tracing cost.

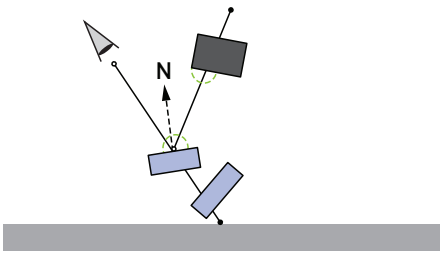

Figure 46-9. *First, discover the transparent surface by tracing a primary ray against transparent objects, and then trace a recursive reflection ray for the closest found surface. The length of the primary ray is limited based on the rasterized opaque depth.*

After discovering a transparent surface, it would be possible to continue the primary ray in order to discover the next transparent surface. However, due to performance reasons, the transparent reflections are limited only to the nearest surface. The cost of the additional reflection rays would grow too high to support more layers in some scenarios.

46.3.2 TRACING TRANSPARENT REFLECTION RAYS

Reflection rays are generated based on what the primary ray hits. One ray per pixel is used as with opaque reflections. The surface normal is evaluated, and the normal map is also applied. However, the possible surface roughness value is ignored, and the reflection ray direction is always evaluated as a perfect mirror reflection without any randomization. This allows avoiding another denoising pass. Most transparent surfaces in the game are actually mirror reflectors, so visually this works without disturbing issues.

As automatic texture LOD level selection is not available in ray tracing, evaluating an approximation for the LOD for sampling the normal map is required. Otherwise, the normal mapped reflection ray directions would be very noisy in some situations. A simple LOD evaluation based on the pixel size in world space proved to be sufficient in this case.

The length of the reflection ray is limited to 60 meters. The limit is not often reached in the game, which contains mostly indoor locations, but it is still applied to keep the performance stable under all scenarios. After the distance limit, the reflections are based on the same environment cube maps that are used when the ray traced reflections are disabled. Near the distance limit, the result is faded from the ray traced result to the cube map–based result to avoid sudden flips between different visual looks.

The shading of the reflection hits is done in the same way as for reflections on opaque surfaces. A separate compute shader pass is executed that applies the simplified and unified shading to the hit surface using the dynamic lights culled by the view-space clustering pass. Even though rough reflections are not supported on transparent surfaces, rich normal map details occasionally lead to very incoherent reflection ray directions, which caused lots of incoherent memory accesses and cache pressure. A specific challenging case is shattered glass, which is fairly common in a shooting game.

46.3.3 ADDING REFLECTIONS TO RASTERIZED TRANSPARENT SURFACES

The evaluated incoming radiance toward a transparent surface from the direction of the reflection ray is not immediately applied to the surface. The actual shading of transparent surfaces happens in a separate rasterization pass. The incoming radiance is stored in a texture along with the depth of the transparent surface. When rasterizing and shading transparent objects, the depth of the shaded surface is compared to the depth value stored in the texture. When they match, the stored radiance is used instead of the environment cube map–based reflection. This approach decouples the shading of the reflection from the shading of the reflecting surface and allows using the same forward shading rasterization approach to render transparent surfaces as is used when ray tracing is disabled.

46.4 NEAR FIELD INDIRECT DIFFUSE ILLUMINATION

The implementation of ray traced indirect diffuse illumination in *Control* resembles the implementation of specular reflections in many ways. However, as using the ray traced indirect illumination also replaces modulating the precomputed global illumination with screen-space occlusion, it has two aspects. It works as ray traced ambient occlusion in addition to actually evaluating dynamic indirect diffuse lighting. The results are shown in Figure 46-10.

Figure 46-10. *Left: precomputed GI is applied on the opaque surface modulated by screen-space occlusion. Right: ray traced dynamic indirect diffuse illumination is applied on the surface in addition to the precomputed GI modulated by ray traced occlusion.*

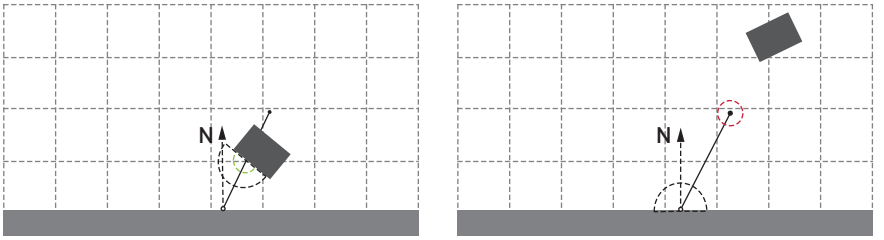

Figure 46-11. *Generated indirect diffuse illumination rays based on the G-buffer. The ray length is limited. When the ray misses, the precomputed GI is applied to the surface instead of the radiance coming from the hit surface. This makes the effect work both as ray traced ambient occlusion and as dynamic indirect diffuse illumination.*

The ray generation happens based on the rasterized G-buffer using cosine distribution for the ray directions as defined by the diffuse Lambert BRDF model. The ray length is limited to one meter. The short rays work well for resolving occlusion as they need to mostly cover only the occlusion from dynamic occluders. The occlusion caused by static large objects at larger distances is already precomputed to the GI data, which is applied to the surface when the ray misses, as illustrated in Figure 46-11. When the ray hits, radiance from the hit surface is applied instead. When evaluating the radiance, the specular BRDF is ignored in order to eliminate noise. The direction-dependent specular highlights could add a considerable amount of noise when the diffuse integral over hemisphere is approximated with only one ray per pixel.

The precomputed GI is used to approximate the second bounce for hits. Similar to the shading of specular reflections, this allows partially unified handling of hits and misses. The GI sampling code is executed regardless of whether the ray hits or misses. And similar to the specular reflections, the shading is executed in a separate compute pass using the simplified, unified material parameterization and shading model and the results of the view-space light clustering pass.

46.5 CONTACT SHADOWS

Traditional shadow maps may suffer from shadow acne due to insufficient resolution or shadow map bias. Though there are ways of mitigating these issues, ray tracing is a low-effort way to not have these problems in the first place.

Figure 46-12. *General overview of contact shadow rendering.*

Though ray traced shadows accurately solve visibility, they might not be a feasible solution for hundreds of lights due to performance reasons, especially if long rays are needed for capturing visibility in a large scene. Because shadow maps are a good and fast solution on a large scale and ray traced shadows excel with short rays, why not use both? Combining the techniques gives great image quality with high performance. Shadow maps handle most cases and ray traced shadows fills in the details. Ray tracing shadows using only very short rays makes them fast and gives the accuracy that we might be missing from shadow maps.

In *Control*, we take the following approach, shown in Figure 46-12: Regular shadow maps are rendered for all shadow casting lights. A few lights are selected for contact shadows and then (non-translucent) visibility is traced for them. The visibility buffer is denoised before it is used in lighting (discussed in Section 46.6.2). Finally, shading is done using both shadow maps and denoised contact shadows.

46.5.1 LIGHT SELECTION

In *Control*, lights are culled using a frustum volume and lighting is deferred. During the main lighting pass, the maximum intensity point lights or spotlights are recorded per pixel. However, not all pixels will be covered by any point lights or spotlights. In that case a pixel is left blank. These lights are ignored in the main lighting, but later added with visibility from both ray traced contact shadows and shadow maps.

Contact shadows are traced for lights, if any, in the maximum intensity buffer. Figure 46-13 shows an example of the maximum intensity light buffer. Black areas indicate that these regions do not have any lighting from point lights or spotlights.

46.5.2 TRACING CONTACT SHADOWS

Tracing shadows is a simple task: trace a ray from the current pixel position in world space toward a selected light. To get soft shadows, the ray direction needs to be jittered with an offset that is within the light's radius.

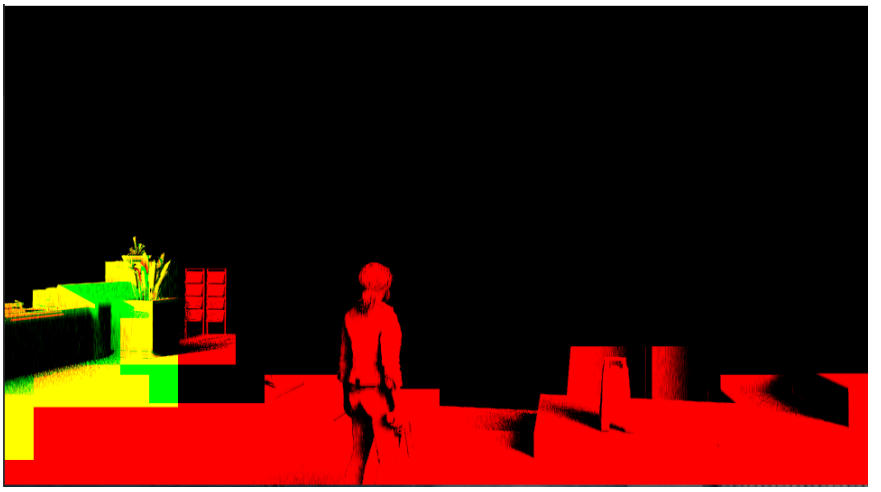

Figure 46-13. *Spotlight index is stored in the red channel and point light index in the green channel. These lights don't always cover the whole screen, and usually there are only a few different lights that dominate.*

Both spotlights and point lights are treated as spherical lights. This will create soft penumbras, but with a cost of noise, which eventually needs denoising before it can be used in lighting. Alternatively, more rays could be traced, but one ray per pixel with a well-designed denoiser and short ray distance gives good results. Denoising of contact shadows is discussed in Section 46.6.2.

The ray direction offset is calculated by first sampling a blue noise texture. This texture gives a random seed that is used to sample a concentric disk. The sample from the disk is multiplied with the light's radius and used to jitter the ray direction using the orthonormal basis of the light.

Contact shadows are only traced for opaque surfaces within the camera frustum. Hit and miss shaders return binary visibility and hit distance (`hitT`), which are both stored. Visibility from both spotlights and point lights are written to the same buffer in separate channels. Compare the results in Figure 46-14.

In *Control*, ray traced shadows are computed only for the two most significant lights, using a limited ray length and a single ray per pixel. This works very well in terms of performance and visual impact.

Figure 46-14. *Left: basic shadow map usage. Right: contact shadows add details on contacts between the floor and other objects.*

46.6 DENOISING

Denoising is an essential part of ray tracing effects. Many ray traced effects are based on Monte Carlo integration, which inherently produce noisy results. Usually, we trade performance for quality, but with a good denoiser we can have both.

Our denoising passes start by implementing a ray generation shader for each ray traced effect. For example, the style of random noise use for jittering the ray direction can affect how the resulting noisy image will look. A poorly chosen noise generator can leave a visible, recognizable pattern in the final image.

Also, the ability to discard sources of noise already while shading the rays can simplify the denoising task. For example, avoiding incoherent rays on rough surfaces and using smooth, precalculated data instead of sparsely sampled rays can result in a cleaner image, which is easier to denoise. However, not all sources of noise are avoidable and different effects may need different filters.

In *Control*, separate denoisers are implemented for reflections, indirect diffuse illumination, and contact shadows.[1] The input data for each of these effects is slightly different and can benefit from different filtering approaches. The filtering approaches used in *Control* were inspired by spatiotemporal variance-guided filtering (SVGF) [5].

46.6.1 DENOISER FOR REFLECTIONS AND INDIRECT DIFFUSE ILLUMINATION

To reduce the sources of noise in the input for reflections, we use shorter rays on rough surfaces, sample precomputed GI data on ray misses, and treat transparent reflections as mirror reflections. The ray direction is jittered with a world-space position–based random noise to diminish screen-space correlations.

Similarly, the ray generation shader for indirect diffuse illumination uses precomputed GI data to minimize noise and the aforementioned scheme for perturbing the ray direction.

The reflection and indirect diffuse illumination denoisers are the most similar and share most of the same code, but are still executed as separate passes. As a first step, they both have a firefly filter (embedded into the temporal pass), which will clamp spiky, high intensities. Firefly clamping is done by sampling the source texture intensity and clamping it with an average luminance from a lower mip level. We use different mip levels and clamping constants for reflections and indirect diffuse illumination.

After intensity clamping, regular temporal accumulation is performed with the previous frame's spatial filtering result, as shown in Listing 46-3. For the reflection denoiser, we apply variance clipping [4].

The temporal filter is followed by a spatial filter. The number of spatial filtering passes depends on the effect. For reflections, the spatial filter is executed twice per direction (horizontally and vertically) and three times for indirect diffuse illumination. The spatial filter samples the current pixel color and takes four samples with an offset (see `extendOffset` in Listing 46-4) that is extended in each pass [5, 6]. Each sample has a weight applied to it, which varies depending on which effect we denoise.

After the weight calculations, all samples are weighted and the final denoised color is resolved. On the last spatial filter iteration, we will temporally

[1] Technically, the reflection and indirect diffuse denoisers could be combined for better performance.

Listing 46-3. *Temporal filter.*

```
void TemporalFilter(...) {
    // Read source: reflections or indirect diffuse illumination.
    float3 finalColor = sourceTexture.Load(uint3(position, 0));
    // Filter high frequencies using lower mip levels of sourceTexture.
    finalColor = clampIntensity(finalColor);
    float2 previousUv = reprojectToPreviousFrame(position);
    float3 previousColor = historyColor.SampleLevel(sampler, previousUv, 0.0
        f);
    float temporalWeight = <user-defined-maximum>;
    temporalWeight *= isDepthValid(currentDepth, previousDepth);
    temporalWeight *= getVelocityWeight(currentUv, previousUv);
#if RELECTIONS
    previousColor = doVarianceClip(previousColor);
#endif
    finalColor = lerp(finalColor, previousColor, temporalWeight);
    targetTexture[position] = finalColor;
}
```

accumulate once more with the result from the first temporal accumulation pass.

After the spatial pass, we add a Fresnel component to the denoised signal—for both reflections and indirect diffuse illumination. Demodulating the Fresnel component gives us a cleaner signal to denoise.

WEIGHTING OPTIONS

Our spatial filter pass uses multiple approaches to weight samples. In addition to temporal accumulation, we use a set of filters depending on the effect. We have tuned these weights for our application. We are mostly validating or weighting input data against various surface attributes and hand-picked constants.

In the reflection denoiser, we use bilateral weights (see Section 46.6.1), filter weights [5], a weight based on hit distance, variance clipping, and a weight based on smoothness. As we are handling rough reflections mostly using precomputed GI data, which reduces input noise, we can bilaterally filter reflections with surface attributes, e.g., depth, normals, and roughness.

The indirect diffuse signal is much noisier than the reflection signal. Thus, we need to take a more relaxed approach in filtering. We mostly want to limit how far we can accept data and not strictly discard it based on surface attributes. The indirect diffuse denoiser uses bilateral weights, filter weights, and a weight based on hit info.

Listing 46-4. *Spatial filter.*

```
void SpatialFilter(...) {

    // Initialize pixel position, pixel UV, previous position, etc.
    ...
    float currentWeight = 1.0f;
    float sampleWeights[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
    float3 currentColor = sourceTexture.Load(position);

    currentWeight *= getWeightsUsingBilateralFilter(...);
    currentWeight *= getWeightsUsingFilterWeights(...);

    for (int i = 0; i < 4; ++i) {   // Samples from neighborhood
        sampleColor[i] = sourceTexture.Load(position + extentOffset(i + 1));
        // Apply a number of filters depending on
        // which effect we are denoising.
        sampleWeights[i] *= getWeightsUsingFilterA(...);
        sampleWeights[i] *= getWeightsUsingFilterB(...);
        sampleWeights[i] *= getWeightsUsingFilterC(...);
        ...
    }

    // Resolve samples.
    currentColor *= currentWeight;
    for (int i = 1; i < 4; ++i) {
        sampleColor[i] *= sampleWeights[i];
    }

    float3 finalColor = currentColor;
    for (int i = 1; i < 4; ++i) {
        finalColor += sampleColor[i];
    }

    // Normalize with total weight from this pass.
    finalColor *= inv(currentWeight + length(sampleWeights));

    // Do one more temporal pass.
    if (isLastIteration) {
        // Same logic as before
        ...
        finalColor = lerp(finalColor, previousColor, temporalWeight);
    }
    targetTexture[position] = finalColor;
}
```

We show denoised results for reflections and indirect diffuse illumination in Figures 46-15 and 46-16, respectively.

BILATERAL WEIGHTS

The bilateral weights are calculated by taking four samples with an offset from depth, normal, smoothness, and material ID buffers. Then, we calculate a weight from each sample set and finally combine these weights into one, which is returned, as shown in Listing 46-5.

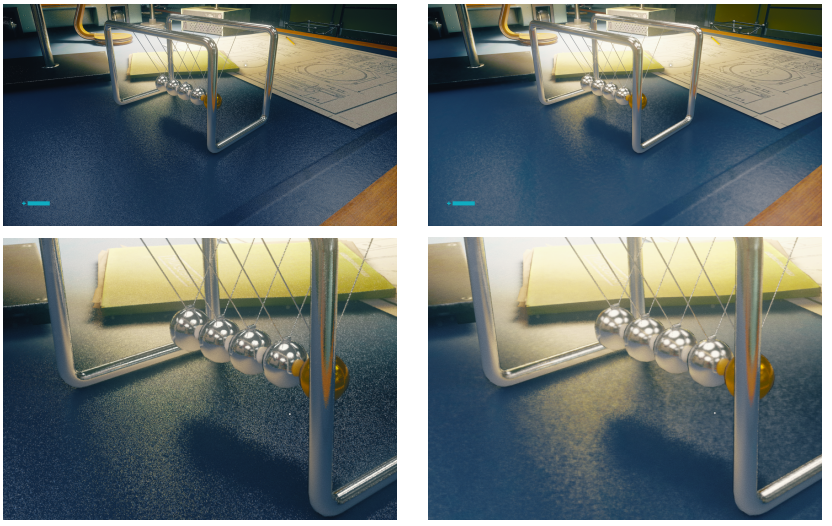

Figure 46-15. *Noisy (left) and denoised (right) reflection buffers compared.*

Figure 46-16. *Noisy (left) and denoised (right) indirect diffuse illumination buffers compared.*

46.6.2 CONTACT SHADOW DENOISER

The contact shadow ray generation shader does two things as a preparation for denoising: it uses blue noise as a random seed for the ray direction offset, and it writes out `hitT` along with the visibility. A random seed is used for sampling a concentric disk. The random sample from the disk is multiplied by the light radius and is used for offsetting the ray direction. The value `hitT` is later used in denoising.

Listing 46-5. *Bilateral weight.*

```
void getBilateralWeight(...) {
    float depth0 = depthBuffer.Load(position);
    float depth1 = depthBuffer.Load(position + offset1);
    float depth2 = depthBuffer.Load(position + offset2);
    ...
    // Repeat for other buffers.
    float4 weightDepth = abs(depth0 - float4(depth1, depth2, depth3, depth4)
        );
    ...
    // Repeat for other sample sets.
    float4 finalWeight = 1.0f;
    finalWeight *= max(<user-defined-min>, saturate(1.0f - weightDepth * <
        user-defined-multiplier>));
    ...
    // Repeat for other weights.
    return finalWeight;
}
```

The shadow denoiser (shown in Listing 46-6) is built similarly to the denoisers for reflections and indirect diffuse illumination. A temporal filter is executed first, which accumulates the previous frame's visibility data with the current frame's visibility if reprojection succeeds. The temporal filter is followed by a spatial filter, which is tailored for shadows: we know which light hit which pixel and can access the information related to that light. That information can be used to denoise shadows efficiently.

Listing 46-6. *Shadow filter.*

```
LightData centerLight; // Fill LightData struct.
...
centerLight.sigma =  GetRadiusInWorld(worldPos, centerLight.worldPosition,
     lightRadius, centerLight.hitT) * 0.6666f;

for (int i = 1; i < filterSize; i += filterStepSize) {
    samplePosition = position + int2(-i,-i) * filterDirection;
    DenoisePixel(denoisedVisibility, sumOfWeights, centerLight,
        samplePosition, currentDepth, positionInWorld);
    samplePosition = position + int2(i, i) * filterDirection;
    DenoisePixel(denoisedVisibility, sumOfWeights, centerLight,
        samplePosition, currentDepth, positionInWorld);
}

denoisedVisibility /= sumOfWeights;
denoisedVisibility = saturate(denoisedVisibility);

// Write out denoisedVisibility.
```

The spatial filter used in *Control* was heavily inspired by the Gameworks spatial shadow filter by Story and Liu [2]. The filter is separable, executed

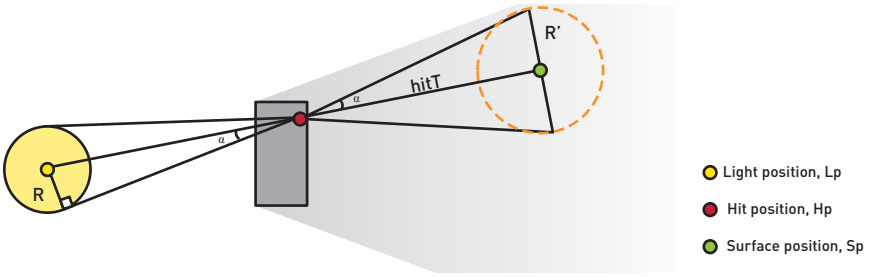

Figure 46-17. *The light radius can be projected onto the shadowed surface using* `hitT`.

once per direction. The idea of the spatial filter is to check per pixel which light potentially created the shadow, approximately project the light radius on the shadowed surface, and use that information to weigh neighboring samples. For sample weighting we used Gaussian weights with sigma calculated from the light radius in world space [3].

First, a filter kernel is initialized retrieving light data for the current pixel, which we refer to as the *center*. We prepare the kernel with the light position in world space, radius, index, visibility, `hitT`, and Gaussian deviation:

```
struct LightData {
    float3  lightPositionInWorld;
    float   hitT;
    float   lightRadius;
    float   sigma;
    float   visibility;
    uint    lightIndex;
};
```

The Gaussian deviation is calculated from the projected light radius (see Figure 46-17). The projected light radius can be calculated using the surface position, light position, and `hitT`. Please refer to Listing 46-7.

Listing 46-7. *Projected light radius.*

```
float GetRadiusInWorld(float3 surfacePosition, float3 lightPosition, float
     lightRadius, float hitT) {
    float3 surfaceToLight = lightPosition - surfacePosition;
    float3 hitPosition = surfacePosition + hitT * normalize(surfaceToLight);
    float hitDistance = length(lightPosition - hitPosition);
    float lightHalfFov = asin(lightRadius / hitDistance);
    return tan(lightHalfFov) * hitT;
}
```

After we've initialized our filter kernel, we can start sampling. We use a kernel radius of eight pixels and take two pixel-wide steps. On each step we call `DenoisePixel`, which samples visibility at the pixel and validates it against the depth and light indices. (See Listing 46-8.) If the visibility data is from the same light as our kernel center, we can potentially use it.

Listing 46-8. *DenoisePixel.*

```
void DenoisePixel(inout float denoisedVisibility, inout float sumOfWeights,
     LightData centerLight, LightData centerPoint, uint2 sampePosition,
     float centerDepth, float3 positionInWorld) {

    // Check point light and spotlight indices at sample pixel.
    uint sampleLightIndex = GetLightIndex(sampePosition);
    float sampleDepth = GetDepth(sampePosition);

    // Sample raw contact shadow buffer.
    float sampleVisibility = GetVisibility(sampePosition);

    // Check sample validity in depth.
    uint isValid = IsValid(sampleDepth, centerDepth) ? 1 : 0;

    float sampleWeight = 0.0f;
    float distanceSampleToCenter = length(positionInWorld -
        samplePositionInWorld);

    // We can use this sample if it's visibility data is
    // from the same light as the center.
    if (sampleLightIndex == centerLight.index && isValid) {
        // Calculate Gaussian weight in world space.
        sampleWeight = GetWeightInWorld(distanceSampleToCenter, centerLight)
            ;
    }

    denoisedVisibility += sampleVisibility * sampleWeight;
    sumOfWeights += sampleWeight;
}
```

We can only use a sample's information if it originates from the same light as the center pixel's sample. In theory, when the maximum intensity light buffer is created, each pixel could get contributions from a different light if there were numerous lights in the scene. In this case, we would not be able to denoise, because all visibility information in each pixel would originate from different lights. Luckily, that was not a common lighting setup for *Control*.

For each valid sample we calculate the sample distance from the kernel center in world space and use it to calculate a Gaussian weight:

```
float GetWeightInWorld(float length, float sigma) {
    if (sigma == 0.0f)
        return (length == 0.0f) ? 1.0f : 0.0f;
    return exp(-(length * length) * rcp(2.0f * sigma * sigma));
}
```

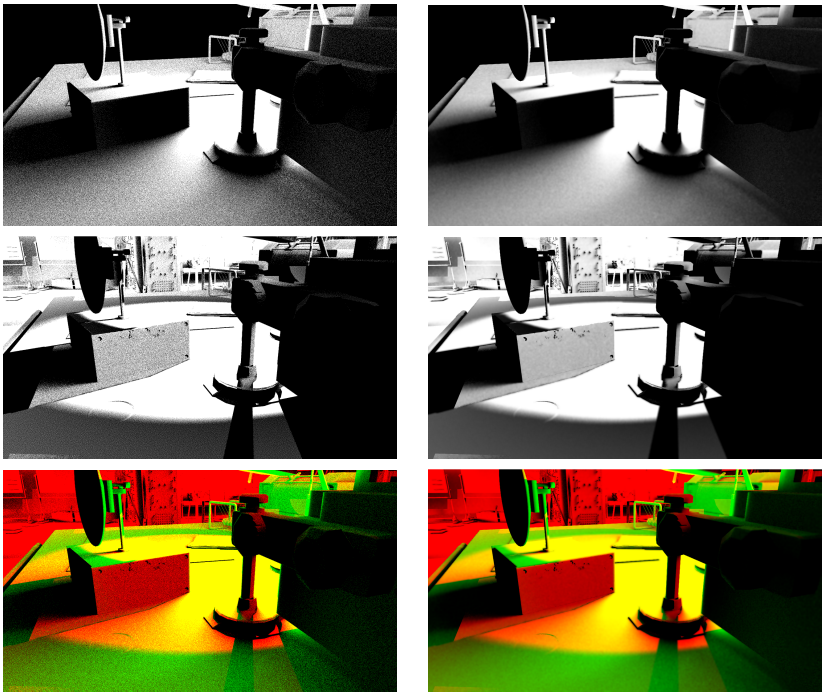

Figure 46-18. *Left: noisy. Right: denoised. Top: spotlight shadow. Middle: shadows from a few point lights. Bottom: combined shadow buffers.*

When all filter steps are done, we write out the denoised visibility and use it in the lighting. Compare the results shown in Figure 46-18.

46.7 PERFORMANCE

Ray tracing performance varies from frame to frame depending on how much of the screen is covered by transparent surfaces and how much of the screen is covered by lights selected for contact shadows. We captured two representative frames at a resolution of 2560×1440 pixels on NVIDIA RTX 3090 to give an example of ray tracing performance. In our example cases, shown in the Figure 46-19, all ray tracing effects take around 8.9 ms of the 19 ms total frame time and 7.1 ms of the 16 ms total. This time is divided into tracing rays, shading the hit results, and denoising. That is a significant fraction of the frame time, but bear in mind that tracing rays enables effects that would not be otherwise possible and adds a lot to the realism of the scene.

Figure 46-19. *Two frames captured at a resolution of* 2560 × 1440 *pixels on NVIDIA RTX 3090 for performance measurements.*

Table 46-1 shows the timing for the individual passes of the ray tracing effects. As we can see, denoising usually takes almost as long as if not even longer than tracing rays. Tracing performance is largely dependent on the ray count and the geometric content of the frame, and denoising only depends on the rendering resolution. Spending an equal amount of time in denoising as in ray tracing might sound a lot, but in practice it offers a good balance between performance and image quality. If we increased the ray count, performance would be impacted significantly. Going from one to two rays per pixel often doubles the time spent in tracing, but likely only has a modest impact on image quality. We argue that a fairly low ray count combined with domain-specific denoisers is the current sweet spot between image quality and performance.

	Frame 1	**Frame 2**
Pass	**Time (ms)**	**Time (ms)**
Acceleration structure building (async.)	0.6	1.0
Reflection ray tracing	1.0	1.4
Reflection shading	1.4	1.1
Reflection denoising	0.8	0.8
Transparent reflection ray tracing	0.8	0.3
Transparent reflection shading	0.7	0.1
Indirect diffuse ray tracing	0.8	0.7
Indirect diffuse shading	0.8	0.6
Indirect diffuse denoising	1.1	1.0
Contact shadow ray tracing	0.8	0.4
Contact shadow denoising	0.7	0.7
Total Cost:	8.9	7.1

Table 46-1. *Frame time spent on different ray tracing effects at a resolution of* 2560 × 1440 *pixels on NVIDIA RTX 3090. The acceleration structure build time is not included in the total cost because it performed asynchronously.*

46.8 CONCLUSIONS

In this chapter, we have shown how ray traced reflections, near field indirect diffuse illumination, and contact shadows can be implemented in a hybrid renderer. By carefully tuning the input signal and designing domain-specific denoisers, we have succeeded in adapting these effect to a visual quality and performance level suitable for a shipped game title. We are very happy with the first launch of the game *Control*. We show that ray tracing can be used to enhance an existing rendering pipeline that is deployed also on platforms without hardware-accelerated ray tracing. Without too much hassle, we were able to bring out visual details and accuracy not possible with traditional rasterization techniques. The effects and their implementation fit comfortably to the game. We look forward to how ray tracing can be utilized in future projects.

REFERENCES

[1] Aalto, T. The latest graphics technology in Remedy's Northlight engine. Presentation at Game Developers Conference, https://www.dropbox.com/s/ni32kzk0twjzwo0/RemedyRenderingGDC18.pptx?dl=0, 2018.

[2] Gruen, H., Roza, M., and Story, J. "Shadows" of the Tomb Raider: Ray tracing deep dive. Presentation at Game Developers Conference, https://www.gdcvault.com/play/1026163/-Shadows-of-the-Tomb, 2019.

[3] Mehta, S., Wang, B., and Ramamoorthi, R. Axis-aligned filtering for interactive sampled soft shadows. *ACM Transactions on Graphics*, 31(6):163:1–163:10, 2012. DOI: 10.1145/2366145.2366182.

[4] Salvi, M. An excursion in temporal supersampling. Presentation at Game Developers Conference, 2016.

[5] 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*, 2:1–2:12, 2017.

[6] Smal, N. and Aizenshtein, M. Real-time global illumination with photon mapping. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, chapter 24, pages 409–436. Apress, 2019.