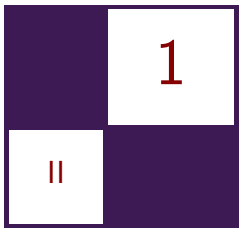

Next-Generation Rendering in Thief

Peter Sikachev, Samuel Delmont, Uriel Doyon, and Jean-Normand Bucci

1.1 Introduction

In this chapter we present the rendering techniques used in *Thief*, which was developed by Eidos Montreal for PC, Playstation 3, Playstation 4, Xbox 360, and Xbox One. Furthermore, we concentrate solely on techniques, developed exclusively for the next-generation platforms, i.e., PC, Playstation 4, and Xbox One.

We provide the reader with implementation details and our experience on a range of rendering methods. In Section 1.2, we discuss our reflection rendering system. We describe each tier of our render strategy as well as final blending and postprocessing.

In Section 1.3, we present a novel contact-hardening shadow (CHS) approach based on the AMD CHS sample. Our method is optimized for Shader Model 5.0 and is capable of rendering high-quality large shadow penumbras at a relatively low cost. Section 1.4 describes our approach toward lit transparent particles rendering.

Compute shaders (CSs) are a relatively new feature in graphics APIs, introduced first in the DirectX 11 API. We have been able to gain substantial benefits for postprocessing using CSs. We expound upon our experience with CSs in Section 1.5.

Performance results are presented in the end of each section. Finally, we conclude and indicate further research directions in Section 1.6.

1.2 Reflections

Reflections rendering has always been a tricky subject for game engines. As long as the majority of games are rasterization based, there is no cheap way to get correct reflections rendered in the most general case. That being said, several methods for real-time reflection rendering produce plausible results in special cases.

1.2.1 Related Methods

One of the first reflection algorithms used for real-time applications, was real-time planar reflection (RTPR) rendering [Lengyel 11]. This method yields an accurate solution for geometry reflected over a plane and is typically used for water or mirror reflections. The method involves rendering objects, or proxies of them, as seen through the reflection plane. Depending on the number of things rendered in the reflection scene, it is possible to balance performance and quality, but the technique is generally considered to be expensive. The main drawback of this method is that in practice, several planes of different heights and orientations would be required to model correctly the environment view surrounding the player, which would be unacceptably expensive to process. This prevents the technique from being used across a wide range of environments.

Cube map reflections are another approach that has been used for many years [NVIDIA Corporation 99]. Though they are very fast and they can handle nonplanar objects, cube maps have their limitations, too. They usually lack resolution and locality compared to other techniques. One also usually needs to precompute cube maps in advance, as it is usually prohibitively expensive to generate cube maps dynamically at runtime. This could additionally complicate an asset pipeline. Precomputed cube maps will not reflect a change in lighting or dynamic objects. Moreover, cube maps do not produce high-quality reflections when applied to planar surfaces, which was one of our main scenarios.

Screen-space reflection (SSR) is a relatively new technique that has grown quickly in popularity [Uludag 14]. It has a moderate performance cost and is easy to integrate. Moreover, it provides great contact reflections (i.e., reflections that occur when an object stands on a reflecting surface; these reflections "ground" an object) hardly achieved by other techniques. However, SSR is prone to numerous artifacts and fails to reflect invisible (or offscreen) parts of a scene. Therefore, it is usually used in combination with some backup technique.

Image-based reflection (IBR) is a method that utilizes planar proxies in order to approximate complex geometries to accelerate ray tracing [Wright 11]. It was developed and shown off in the Unreal Engine 3 *Samaritan* demo. IBR can achieve good results in reflection locality and allows an arbitrary orientation of a reflector. However, the complexity grows linearly with the number of proxies, which could become prohibitive for large scenes.

Figure 1.1. From left to right: cube map reflections only, SSR + cube maps, IBR + cube maps, and SSR + IBR + cube maps. [Image courtesy Square Enix Ltd.]

Numerous variations of the methods discussed above have been proposed and used in real-time rendering. For instance, localized, or parallax-corrected cube maps [Lagarde and Zanuttini 13] are arguably becoming an industry standard. In the next sections, we will describe the reflection system we used in *Thief*.

1.2.2 Thief Reflection System Overview

Creating a reflection system that perfectly handles every surface type, in real time, is a very difficult problem. Therefore, together with the art department, we developed a specification of requirements and limitations for the *Thief* reflection system. Given that *Thief* was originally designed for the Xbox 360 and Playstation 3 generation of platforms, we had quite a generous performance budget for the reflection system on the next-generation platforms: 5 ms. Beforehand, we implemented the real-time planar reflection method, which ran at 10 ms. This running time was obviously unacceptable; moreover, this technique could render reflections for only one plane.

The majority of reflections in the game world come from the ground (wet spots, tile, etc.), therefore we limited ourselves to quasi-horizontal surfaces. However, since *Thief* is a multilevel game (e.g., you can make your way across rooftops instead of streets), unlike [Lagarde and Zanuttini 13], we could not be limited to a single reflection plane. As mentioned above, we performed tests with PRTR and the single reflection plane limitation was insufficient for our assets.

The target was to accurately capture human-sized objects and contact reflections. In addition, we also wanted to capture principal landmarks (e.g., large buildings). Finally, as torches and bonfires are a typical light source in the world of *Thief*, we needed a way to render reflection from certain transparent geometry as well.

To achieve these goals, we came up with a multitier reflection system, outlined in Figure 1.1. The reflection system of *Thief* consists of the following tiers:

- screen-space reflections (SSR) for opaque objects, dynamic and static, within a human height of a reflecting surface;
- image-based reflections (IBR) for walls and far away landmarks;
- localized cube map reflections to fill the gaps between IBR proxies;
- global cube map reflections, which are mostly for view-independent skyboxes.

Each tier serves as a fallback solution to the previous one. First, SSR ray-marches the depth buffer. If it does not have sufficient information to shade a fragment (i.e., the reflected ray is obscured by some foreground object), it falls back to image-based reflection. If none of the IBR proxies are intersected by the reflection ray, the localized cube map reflection system comes into play. Finally, if no appropriate localized cube map is in proximity, the global cube map is fetched. Transition between different tiers is done via smooth blending, as described in Section 1.2.6.

1.2.3 Screen-Space Reflections

SSR is an image-based reflection technique based on ray-marching through the depth buffer [Kasyan et al. 11]. We use the lit color buffer, the normal buffer, and the depth buffer from the current frame. SSR is applied before rendering translucent geometry in order to avoid perspective artifacts.

At each fragment we reconstruct the camera-space position, using the screen uv-coordinates, the fetched depth, and the projection matrix. Afterward, we ray-march with a constant step along the reflected ray until the analytical depth of the fetch along the ray is more than the depth buffer fetch from the same screen-space position. Finally, the intersection location is refined with several binary search steps, as shown in Figure 1.2.

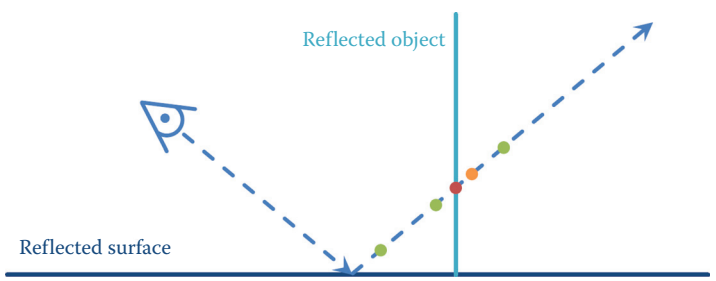

Figure 1.2. Screen-space reflections linear steps (green) and binary search steps (orange and then red).

This method yields very accurate reflections at the contact of a reflective surface and a reflected object. There are, however, several major issues with SSR. First, this method is very expensive: at each fragment we need to make several potentially uncoalesced (due to the discrepancy of reflection rays) texture fetches, some of which are even view-dependent (binary search). Second, reflected information is often missing: it's either out of screen or obscured by a closer object.

We address the first issue with several optimization techniques, described below in this subsection. The second issue is addressed by falling back to the subsequent tier of the reflection system.

We decrease the memory traffic by approximating the normal at the fragment with a normal pointing vertically up. This dramatically increases the number of texture fetches at neighboring fragments that might be coalesced and processed within a single dynamic random-access memory (DRAM) burst. However, this naturally results in ideal mirror-like reflections without any normal perturbation. We address this issue further in Section 1.2.7.

Moreover, we use dynamic branching to employ early exit for the constant step loop when the first intersection with the depth buffer is found. Although it might result in false ray-depth buffer collision detection, we compromise on accuracy in order to further save on bandwidth.

Another optimization is decreasing the number of samples for the distant fragments. We came up with an empirical formula that decreases the number of steps proportional to the exponent of the distance:

$$N_{\text{linear samples}} = \max(1, k_1 e^{-k_2 d}),$$

where depth is denoted with d, k_1 is the linear factor, and k_2 is the exponential factor.

Additionally, we use a bunch of early-outs for the whole shader. We check if the surface has a reflective component and if a reflection vector points to the camera. The latter optimization does not significantly deteriorate visual quality, as in these situations SSR rarely yields high-quality results anyway and the reflection factor due to the Fresnel equation is already low. Moreover, this reduces the SSR GPU time in the case when IBR GPU time is high, thus balancing the total.

However, one should be very careful when implementing such an optimization. All fetches inside the `if`-clause should be done with a forced mipmap level; all variables used after should be initialized with a meaningful default value, and the `if`-clause should be preceded with a `[branch]` directive. The reason is that a shader compiler might otherwise try to generate a gradient-requiring instruction (i.e., `tex2D`) and, therefore, flatten a branch, making the optimizations useless.

1.2.4 Image-Based Reflections

Image-based reflections are a reflection technique implemented in [Wright 11]. The key idea is to introduce one or more planar quad reflection proxies and pre-render an object of interest into it. During fragment shading, we just ray-trace the reflected ray against an array of proxies in the pixel shader. A similar idea is utilized in [Lagarde and Zanuttini 13]. However, in the latter, the reflections are rendered only for planes at a single height. Therefore, we were unable to use optimizations proposed in [Lagarde and Zanuttini 13].

In *Thief*, our ambition was to have around 50 IBR proxies per scene. IBR back-face culling effectively reduces the visible proxy count by half. A straightforward approach resulted in well over 8 ms of GPU time at the target configuration, which was totally unacceptable. Therefore, we employed a series of acceleration techniques, described below.

First, we utilized the same approach for normal approximation as for SSR to increase memory coalescing. This allowed the following optimizations:

- rejecting planes not facing the player,
- rejecting planes behind the player,
- tile-based IBR rendering (discussed below).

Bump perturbation was then performed for SSR and IBR together.

Second, we introduced tile-based IBR. Because we have limited ourselves to quasi-horizontal reflections, we divided the entire screen space into a number of vertical tiles. Our experiments have proven 16 tiles to be an optimal number. After that, for each reflection proxy, we calculate the screen-space coordinates for each vertex. If a vertex is in front of the near clip plane, we flip the w sign before perspective divide in order to handle close proxies. Then x-coordinates of the transformed vertices might be used as minimum and maximum values to determine the tiles covered by the proxy's reflection.

However, due to perspective projection, this method would result in reflections being cut, especially when a proxy approaches the screen borders. To fix that, we introduce the following workaround. For each of two vertical sides of the proxy, we extend them to the intersections with top and bottom screen borders as shown in Figure 1.3. The resultant x-coordinates are used to decrease the minimum and/or increase the maximum. The pseudocode of this method is shown in Algorithm 1.1.

The above mentioned optimization decreases GPU time dramatically; however, if a player looks straight down, all the proxies start occupying almost all tiles due to high perspective distortions. To alleviate performance drops in this case, we use a bounding sphere test in the pixel shader for an early-out before the actual high-cost tracing. While this check deteriorates the performance in the most common cases, it increases GPU performance in the worst cases, resulting in a more consistent frame rate.

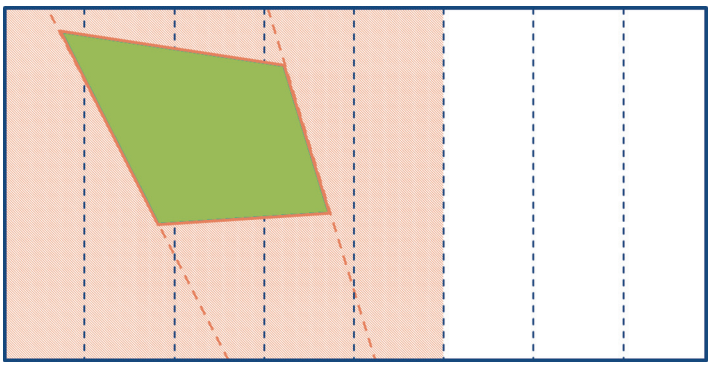

Figure 1.3. IBR tile-based rendering optimization. IBR proxy is shown in orange. Tiles are shown in dotted blue lines, and vertical sides extensions are shown in orange dotted lines. The affected tiles are shaded with thin red diagonal lines.

```
1: x_min = 1
2: x_max = −1
3: for all IBR proxies in front of player and facing player do
4:    find AABB of the current proxy
5:    for all vertices of AABB do
6:       calculate vertex coordinate in homogeneous clip space
7:       w := |w|
8:       calculate vertex coordinate in screen clip space
9:       x_min := min(x, x_min)
10:      x_max := max(x, x_max)
11:   end for
12:   for all vertical edges of AABB in screen space do
13:      calculate intersections x_1 and x_2 with top and bottom of the screen
14:      x_min := min(x_1, x_2, x_min)
15:      x_max := max(x_1, x_2, x_max)
16:   end for
17:   for all IBR tiles do
18:      if the tile overlaps with [x_min, x_max] then
19:         add the proxy to the tile
20:      end if
21:   end for
22: end for
```

Algorithm 1.1. Algorithm for finding affected tiles for an IBR proxy.

Additionally, in order to limit the number of active IBR proxies in the frame, we introduced the notion of *IBR rooms*. Essentially, an IBR room defines an AABB so that a player can see IBR reflections only from the IBR proxies in

Figure 1.4. Non-glossy reflection rendering (left) and CHGR (right). [Image courtesy Square Enix Ltd.]

the same room. Moreover, the lower plane of an IBR room's AABB defines the maximum *reflection extension* of each of the proxies inside it. This allowed us to drastically limit the number of reflections when a player is looking down.

As a side note, *Thief* has a very dynamic lighting environment. In order to keep the IBR reflection in sync with the dynamic lights, IBR had to be scaled down based on the light intensity. This makes the IBR planes disappear from reflection when lights are turned off. Although this is inaccurate since the IBR textures are generated from the default lighting setup, it was not possible to know which parts of the plane were actually affected by dynamic lighting.

Also, IBRs were captured with particles and fog disabled. Important particles, like fire effects, were simulated with their own IBRs. Fog was added accordingly to the fog settings and the reflection distance after blending SSR and IBR.

1.2.5 Contact-Hardening Glossy Reflections

Because the majority of the reflecting surfaces in *Thief* are not perfect mirror reflectors, we decided to simulate glossy reflections. Glossy SSR reflections are not a new feature, having been first implemented in [Andreev 13]. We decided to take SSR a step further and render contact-hardening glossy reflections (CHGRs). An example of a CHGR is shown in Figure 1.4.

The main phenomena we wish to capture is that a reflection is sharpest near the contact point of the reflected object and the reflecting surface. The reflection grows more blurry as these two surfaces get farther away from each other.

The algorithm for CHGR rendering is as follows. First, we output the distance between the reflecting surface and the point where the reflected ray hits the reflected object. Because we want to limit the size of the render targets, we

```
//World-space unit is 1 centimeter
int distanceLo = int(worldSpaceDistance) % 256;
int distanceHi = int(worldSpaceDistance) / 256;

packedDistance = float2(float(distanceLo) / 255.0f,
                        float(distanceHi) / 255.0f);
```

Listing 1.1. Reflection distance packing.

```
float3 reflectedCameraToWorld =
    reflect(cameraToWorld, worldSpaceNormal);
float reflectionVectorLength =
    max(length(reflectedCameraToWorld), FP_EPSILON);
float worldSpaceDistance = 255.0f * (packedDistance.x +
                           256.0f *  packedDistance.y) /
                           reflectionVectorLength;
...
//Reflection sorting and blending
...
float4 screenSpaceReflectedPosition =
    mul(float4(reflectedPosition, 1), worldToScreen);
screenSpaceReflectedPosition /= screenSpaceReflectedPosition.w;

ReflectionDistance = length(screenSpaceReflectedPosition.xy -
                            screenSpaceFragmentPosition.xy);
```

Listing 1.2. Reflection distance unpacking.

utilize R8G8B8A8 textures for color and depth information. As 8 bits does not provide enough precision for distance, we pack the distance in two 8-bit channels during the SSR pass, as shown in Listing 1.1.

The IBR pass unpacks the depth, performs blending, and then converts this world-space distance into screen-space distance as shown in Listing 1.2. The reason for this is twofold. First, the screen-space distance fits naturally into the $[0, 1]$ domain. As we do not need much precision for the blurring itself, we can re-pack it into a single 8-bit value, ensuring a natural blending. Second, the screen-space distance provides a better cue for blur ratio: the fragments farther away from the viewer should be blurred less than closer ones, if both have the same reflection distance.

The second step is to dilate the distance information. For each region, we select the maximum distance of all the pixels covered by the area of our blur kernel. The reason for this is that the distance value can change suddenly from one pixel to the next (e.g., when a close reflection proxy meets a distant background pixel). We wish to blur these areas with the maximum blur coefficient from the corresponding area. This helps avoid sharp silhouettes of otherwise blurry objects. This problem is very similar to issues encountered with common depth-

of-field rendering algorithms. In order to save on memory bandwidth, we apply a two-pass, separable dilation maximum filter. This provides us with an acceptable approximation.

Finally, we perform the blur with the adjustable separable kernel. In addition to selecting the Gaussian parameters based on the distance value, we also apply the following tricks. First, we ignore the samples with zero specular intensity in order to avoid bleeding at the silhouette of an object. This requires on-the-fly adjustment of the kernel in the shader. Second, we follow the same heuristic as in [Andersson 13], so we blur the image more in the vertical direction than in the horizontal direction, achieving more plausible visual results.

1.2.6 Reflection Blending

As our reflection system consists of several tiers, we need to define how we blend between them. In addition to the distance factor, our SSR pass also outputs a blending factor. This factor depends on the following:

- height (the longer we cast a reflection ray, the less contribution it makes),
- tracing accuracy (depth delta between the ray coordinate and the fetched depth),
- surface tilt (the more the surface normal diverges from vertical, the less SSR should contribute),
- Reflection ray going toward camera or out of screen.

Afterward, IBR is merged on top, outputting a cumulative blending factor. Finally, the cube map is applied. Figure 1.5 shows seamless blending between SSR and IBR.

Figure 1.5. SSR only (left) and SSR blended with IBR (right). [Image courtesy Square Enix Ltd.]

Figure 1.6. Reflection blending without sorting (left) and with sorting (right). [Image courtesy Square Enix Ltd.]

However, this approach causes certain problems in cases when a transparent IBR proxy is in front of the object that could be potentially reflected with SSR. Figure 1.6 shows the issue. To address this problem, instead of simply blending SSR and IBR, we perform layer sorting beforehand. We create a small (three to four entries) array of reflection layers in the IBR shader and inject SSR results into it as the first element. The array is kept sorted when we add every subsequent IBR trace result. Thus, we end up with the closest intersections only.

1.2.7 Bump as a Postprocess

As mentioned above, we assume that the normal is pointing up for SSR and IBR rendering in order to apply acceleration techniques. Furthermore, in order to reduce memory bandwidth, we render reflection at half resolution. Together, this diminishes high-frequency details, which are crucial for reflections, especially on highly bumped surfaces. To alleviate this, we apply a *bump* effect as a postprocess when upscaling the reflection buffer to full resolution.

The main idea is very similar to the generic refraction approach [Sousa 05]. We use the difference between the vertical normal and the per-pixel normal to offset the UV in the rendered reflection texture. To fight reflection leaking, we revert to the original fetch if the new fetch is significantly closer than the old one. Figure 1.7 shows the benefits of applying reflection bump.

1.2.8 Localized Cube Map Reflections

In the SSR, IBR, and cube map reflection strategies, the cube map would ideally only contain the skybox and some far away geometry since the playable environment would be mapped by IBR planes. In this situation, only one cube map would be required. In practice, the IBR planes have many holes and do not connect perfectly to each other. This is a consequence of how our IBR planes are generated using renderable textures.

Figure 1.7. Reflection without bump (left) and with bump as a postprocess (right). [Image courtesy Square Enix Ltd.]

When a reflected ray enters into one of these cracks and hits the skybox, it results in a bright contrast pixel because most *Thief* scenes typically use a skybox that is much brighter than the rest of the environment. To fix this, we used localized cube maps taken along the playable path. Any primitive within reach of a localized cube map would then use it in the main render pass as the reflected environment color.

Technically, the cube map could be mapped and applied in screen space using cube map render volumes, but we chose to simply output the cube map sample into a dedicated render target. This made the cube map material-bound and removed its dependency with the localized cube map mapping system.

The main render pass in *Thief* would output the following data for reflective primitives:

- material lit color (sRGB8),
- environment reflection color + diffuse lighting intensity (sRGB8),
- world normal + reflectivity (RGB8).

After generating the IBR and SSR half-resolution reflection texture, the final color is computed by adding SSR, IBR, and finally the environment reflection color (i.e., cube map color). If the material or platform does not support IBR/SSR, the color would simply be added to the material-lit color and the extra render targets are not needed.

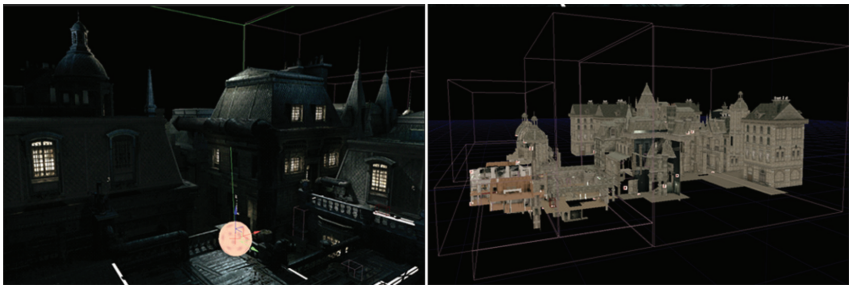

Figure 1.8. Creation of the multiple volumes for covering the whole environment. [Image courtesy Square Enix Ltd.]

Note here that we needed the diffuse lighting intensity to additionally scale down the IBR and cube map color because they were captured with the default lighting setup, which could be very different from the current in-game lighting setup. This scale was not required for the SSR because it is real-time and accurate, while the IBR proxies and cube maps are precomputed offline.

1.2.9 Art Pipeline Implications

For each level in *Thief*, we set a default cube map. This one is used by all the geometry that has reflective materials. This cube map is pretty generic and reflects the most common area of the level.

We then identify the areas where we require a more precise cube map definition. The decision is often based on lighting conditions or the presence of water puddles. For an area that could benefit from a more precise cube map, artists add a localized cube map, which comes with its own scene capture preview sphere. This is shown in the Figure 1.8.

We then built our entire environment using these volumes that enhance the look of the level. After using a build process, we generated all the cube maps for the volumes created. Figure 1.9 shows the additional build process created with the different resources generated.

Thief's reflection systems strongly enhanced the visual quality and next-generation look of the game. It was deeply embedded into the artistic direction of the title. The water puddles, in particular, helped create the unique look our artists wanted for the game. This also contributed to the gameplay of *Thief*. Because light and shadows are so important to our gameplay, the reflective water puddles became an additional challenge the player must manage when trying to stay hidden in the shadows.

For *Thief*, we dedicated a budget of 5 ms for the reflection pipeline. Given the allocated budget and the production stage at which we pushed the data in, we had to make clever choices, sometimes using all available techniques and other times

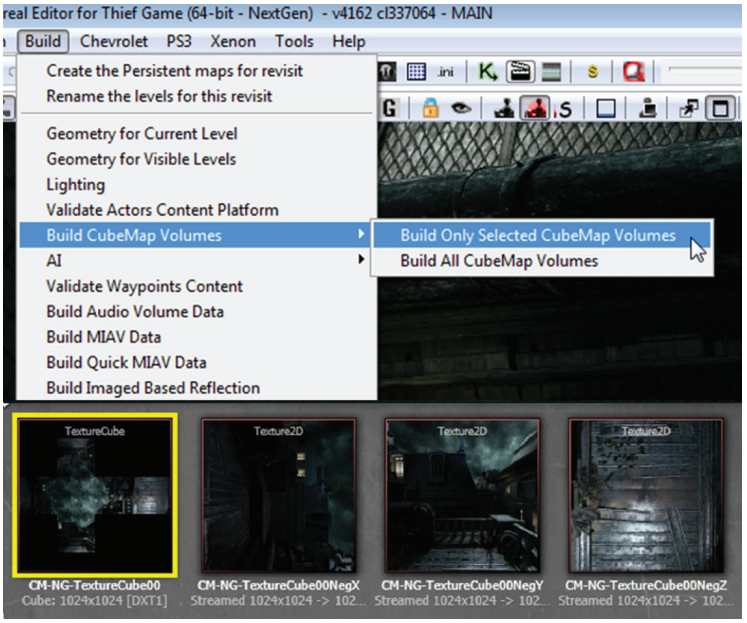

Figure 1.9. Build process generating the individual cube map captures. [Image courtesy Square Enix Ltd.]

using only one, given the rendering stresses of an environment. For example, certain environment condition might force us to use only one technique. Poor lighting condition could be a good example where you do not want to pay the extra cost of an expensive cube map or IBR planes.

We found that screen-space reflections were very easy for our art team to integrate. For this reason, we used SSR as our base tool for most of our reflection needs. This would then dictate where some of our IBR planes should go; it was a fallback solution when SSR failed.

1.2.10 Results

We came up with a robust and fast reflection system that is ready for the next-generation consoles. Both SSR and IBR steps take around 1–1.5 ms on Playstation 4 (1080p) and Xbox One (900p). However, these are worst case results, i.e., taken on a synthetic scene with an SSR surface taking up the whole screen and 50 IBR proxies visible. For a typical game scene, the numbers are usually lower than that. Reflection postprocessing is fairly expensive (around 2 ms). However, we did not have time to implement it using compute shaders, which could potentially save a lot of bandwidth.

Figure 1.10. Shadow with ordinary filtering (left) and with contact-hardening shadows (right). [Image courtesy Square Enix Ltd.]

Our reflection system does not support rough reflections. Taking into account the emerging interest in physically based rendering solutions, we are looking into removing this limitation. Reprojection techniques also look appealing both for quality enhancement and bandwidth reduction.

1.3 Contact-Hardening Shadows

Contact-hardening shadows (CHSs), similar to percentage-closer soft shadows (PCSSs) [Fernando 05], are a shadow-mapping method to simulate the dynamic shadows from area lights. The achieved effect is a sharper shadow as the shadow caster and the receiver are closer to each other and a blurrier (softer) shadow as the caster and the receiver are farther from each other (see Figure 1.10). The implementation in *Thief* is based on the method from the AMD SDK [Gruen 10].

This method is easy to integrate because it uses the shadow map generated by a single light source and can just replace ordinary shadow filtering. One of the main drawbacks in this technique is the extensive texture fetching, and in *Thief* we implemented an optimized method for Shader Model 5.0 that drastically limits the access to the shadow map. The CHS process is divided into three steps, which are the blocker search, the penumbra estimation, and the filtering.

1.3.1 Blocker Search

The first step consists of computing the average depth of the blockers inside a search region around the shaded point (that we will reference as *average blocker depth*). A kernel grid of $N \times N$ texels centered at the shaded point covers this search region. A blocker is a texel in the shadow map representing a point closer to the light than the currently computed fragment, as shown in Figure 1.11. This average-blocker-depth value will be used in the penumbra estimation.

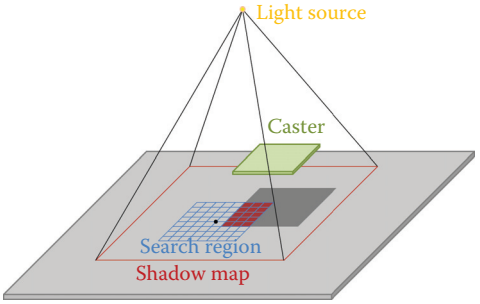

Figure 1.11. Blockers in a 8×8 search grid.

Shader Model 5.0's new intrinsic `GatherRed()` accelerates this step by sampling four values at once. In *Thief*, we decided to use a 8×8 kernel size, which actually performs 16 samples instead of 64 for a Shader Model 4.0 implementation (see Listing 1.4). Increasing the size of the kernel will allow a larger penumbra, since points that are farther from the shaded one can be tested, but it obviously increases the cost as the number of texture fetches grows.

Because the penumbra width (or blurriness) is tightly related to the size of the kernel, which depends on the shadow map resolution and its projection in world space, this leads to inconsistent and variable penumbra width when the shadow map resolution or the shadow frustum's FOV changes for the same light caster/receiver setup. Figure 1.12 shows the issue.

To fix this issue in *Thief*, we extended the CHS by generating mips for the shadow map in a prepass before the CHS application by downsizing it iteratively. Those downsizing operations are accelerated with the use of the `GatherRed()` intrinsic as well. Then, in the CHS step, we dynamically chose the mip that gives

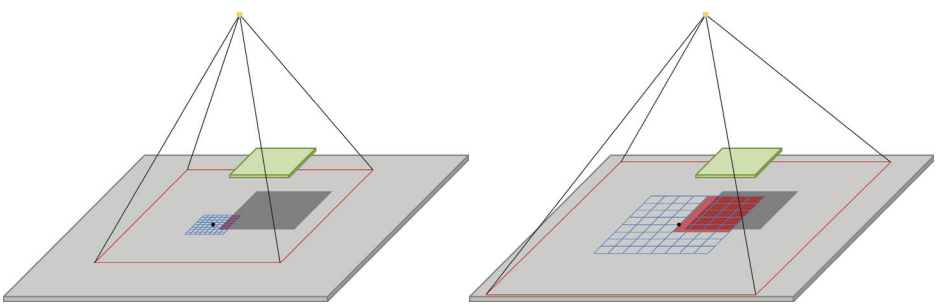

Figure 1.12. For the same 8×8 search grid, a smaller search region due to higher resolution shadow map (left) and a bigger search region due to wider shadow frustum (right).

```
#define KERNEL_SIZE 8
float wantedTexelSizeAt1UnitDist =
  wantedPenumbraWidthAt1UnitDist / KERNEL_SIZE;
float texelSizeAt1UnitDist =
  2*TanFOVSemiAngle/shadowMapResolution;
float MaxShadowMip =
  -log(texelSizeAt1UnitDist/wantedTexelSizeAt1UnitDist)/log(2);
MaxShadowMip = min(float(MIPS_COUNT-1),max(MaxShadowMip,0.0));
// both BlkSearchShadowMipIndex and MaxShadowMip are passed
// to the shader as parameters
int BlkSearchShadowMipIndex = ceil(MaxShadowMip);
```

Listing 1.3. Algorithm for choosing a mip from a user-defined penumbra width, the shadow map resolution, and the FOV angle of the shadow frustum.

Figure 1.13. Shadow-map mips layout. [Image courtesy Square Enix Ltd.]

a kernel size in world space that is closer to a user-defined parameter. Listing 1.3 shows how the mip index is computed from this user-defined parameter, the shadow map resolution, and the FOV angle of the shadow frustum. This process can be done on the CPU and the result is passed to a shader as a parameter.

Unfortunately, the `GatherRed()` intrinsic does not allow mip selection. Therefore, the mips are stored in an atlas, as shown in Figure 1.13, and we offset the texture coordinates to sample the desired mip. This is achieved by applying a simple offset scale to the coordinates in texture space (see Listing 1.4).

In order to save on fragment instructions, the function returns, as an early out, a value of 0.0 (fully shadowed) if the average blocker depth is equal to 1.0 (found a blocker for all samples in the search region) or returns 1.0 (fully lit) if the average blocker depth is equal to 0.0 (no blocker found). Listing 1.4 shows the details of the average-blocker-depth compute.

```
#define KERNEL_SIZE 8
#define BFS2 (KERNEL_SIZE - 1) / 2

float3 blkTc = float3(inTc.xy, inDepth);
// TcBiasScale is a static array holding the offset-scale
in the shadow map for every mips.
float4 blkTcBS = TcBiasScale[BlkSearchShadowMipIndex];
blkTc.xy = blkTcBS.xy + blkTc.xy * blkTcBS.zw;
// g_vShadowMapDims.xy is the shadow map resolution
// g_vShadowMapDims.zw is the shadow map texel size
float2 blkAbsTc = ( g_vShadowMapDims.xy * blkTc.xy );
float2 fc = blkAbsTc - floor( blkAbsTc );
blkTc.xy  = blkTc.xy - ( fc * g_vShadowMapDims.zw );
float blkCount = 0; float avgBlockerDepth = 0;
[loop]for( int row = -BFS2; row <= BFS2; row += 2 )
{
  float2 tc = blkTc.xy + float2(-BFS2*g_vShadowMapDims.z,
              row*g_vShadowMapDimensions.w);
  [unroll]for( int col = -BFS2; col <= BFS2; col += 2 )
  {
    float4 depth4 = shadowTex.GatherRed(pointSampler, tc.xy);
    float4 blk4 = (blkTc.zzzz <= depth4)?(0).xxxx:(1).xxxx;
    float4 fcVec = 0;
    if (row == -BFS2)
    {
      if (col == -BFS2)
        fcVec = float4((1.0-fc.y) * (1.0-fc.x),
                       (1.0-fc.y), 1, (1.0-fc.x));
      else if (col == BFS2)
        fcVec = float4((1.0-fc.y), (1.0-fc.y) * fc.x, fc.x, 1);
      else
        fcVec = float4((1.0-fc.y), (1.0-fc.y), 1, 1);
    }
    else if (row == BFS2)
    {
      if (col == -BFS2)
        fcVec = float4((1.0-fc.x), 1, fc.y, (1.0-fc.x) * fc.y);
      else if (col == BFS2)
        fcVec = float4(1, fc.x, fc.x * fc.y, fc.y);
      else
        fcVec = float4(1, 1, fc.y, fc.y);
    }
    else
    {
      if (col == -BFS2)
        fcVec = float4((1.0-fc.x), 1, 1, (1.0-fc.x));
      else if (col == BFS2)
        fcVec = float4(1, fc.x, fc.x, 1);
      else
        fcVec = float4(1,1,1,1);
    }
    blkCount += dot(blk4, fcVec.xyzw);
    avgBlockerDepth += dot(depth4, fcVec.xyzw*blk4);
    tc.x += 2.0*g_vShadowMapDims.z;
  }
}
if( blkCount == 0.0 ) // Early out - fully lit
  return 1.0f;
else if (blkCount == KERNEL_SIZE*KERNEL_SIZE) // Fully shadowed
  return 0.0f;
avgBlockerDepth /= blkCount;
```

Listing 1.4. Average-blocker-depth compute.

$$\text{width}_{\text{penumbra}} = \frac{(\text{depth}_{\text{receiver}} - \texttt{avgBlockerDepth}) \cdot \text{width}_{\text{light}}}{\texttt{avgBlockerDepth}}$$

Algorithm 1.2. Algorithm for computing the penumbra estimation.

1.3.2 Penumbra Estimation

Based on the average-blocker-depth value from the previous step and the user-defined light width, a factor (the penumbra estimation) is computed. Algorithm 1.2 is pretty straightforward and is the same as many other PCSS implementation.

1.3.3 Filtering

The final CHS step consists of applying a dynamic filter to the shadow map to obtain the light attenuation term. In this step, we also take advantage of the shadow-map mips. The main idea is to use higher-resolution mips for the sharp area of the shadow and lower-resolution mips for the blurry area. In order to have a continuous and unnoticeable transition between the different mips, we use two mips selected from the penumbra estimation and perform one filter operation for each mip before linearly blending the two results (see Figure 1.14). Doing so

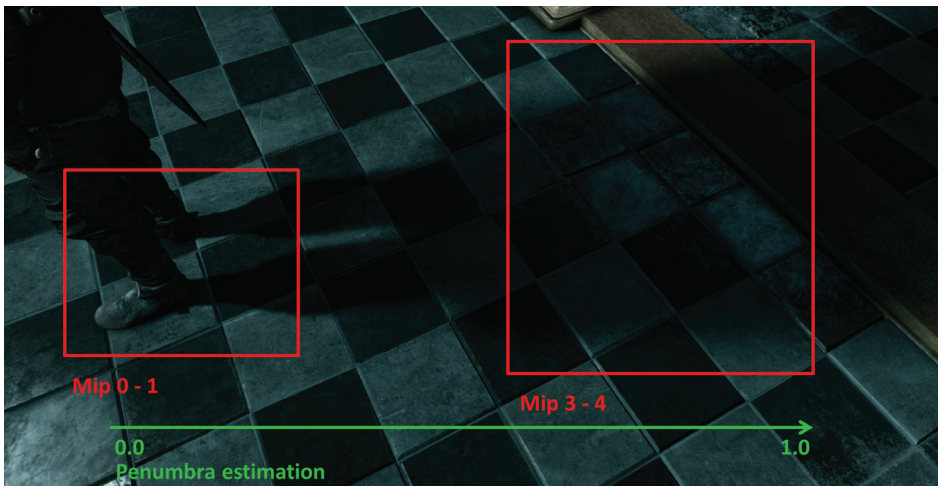

Figure 1.14. Mips used for the filtering, depending on the user-defined region search width and the penumbra estimation. [Image courtesy Square Enix Ltd.]

```
#define KERNEL_SIZE 8
#define FS2 (KERNEL_SIZE - 1) / 2

float Ratio = penumbraWidth;
float clampedTexRatio = max(MaxShadowMip - 0.001, 0.0);
float texRatio = min(MaxShadowMip * Ratio, clampedTexRatio);
float texRatioFc = texRatio - floor(texRatio);
uint textureIndex = min(uint(texRatio), MIPS_COUNT-2);
float4 highMipTcBS = TcBiasScale[textureIndex]; // higher res
float4 lowMipTcBS  = TcBiasScale[textureIndex+1]; // lower res
// Pack mips Tc into a float4, xy for high mip, zw for low mip
float4 MipsTc = float4(highMipTcBS.xy + inTc.xy*highMipTcBS.zw,
                       lowMipTcBS.xy + inTc.xy*lowMipTcBS.zw);
float4 MipsAbsTc = (g_vShadowMapDims.xyxy * MipsTc);
float4 MipsFc = MipsAbsTc - floor(MipsAbsTc);
MipsTc  = MipsTc - (MipsFc * g_vShadowMapDims.zwzw);
...
//Apply the same dynamic weight matrix to both mips
//using ratio along with the corresponding MipsTc and MipsFc
...
return lerp(highMipTerm, lowMipTerm, texRatioFc);
```

Listing 1.5. Shadow mips filtering and blending.

gives a realistic effect with variable levels of blurriness, using the same kernel size (8×8 in *Thief*) through the whole filtering. The highest mip index possible (which corresponds to a penumbra estimation of 1.0) is the same one used in the blocker search step.

As described above, we need to get the attenuation terms for both selected mips before blending them. A dynamic weight matrix is computed by feeding four matrices into a cubic Bézier function, depending only on the penumbra estimation, and used to filter each mip (not covered here; see [Gruen 10] for the details). Like the previous steps, this is accelerated using the `GatherCmpRed()` intrinsic [Gruen and Story 09]. Listing 1.5 shows how to blend the filtered mips to obtain the final shadow attenuation term.

The number of shadow map accesses for the blocker search is 16 (8×8 kernel with the use of `GatherCmpRed()`) and 2×16 for the filter step (8×8 kernel for each mip with the use of `GatherCmpRed()`), for a total of 48 texture fetches, producing very large penumbras that are independent from the shadow resolution (though the sharp areas still are dependent). A classic implementation in Shader Model 4.0 using a 8×8 kernel with no shadow mipmapping would perform 128 accesses for smaller penumbras, depending on the shadow resolution.

Performance-wise, on an NVIDIA 770 GTX and for a 1080p resolution, the CHS takes 1–2 ms depending on the shadow exposure on the screen and the shadow map resolution. The worst case corresponds to a shadow covering the whole screen.

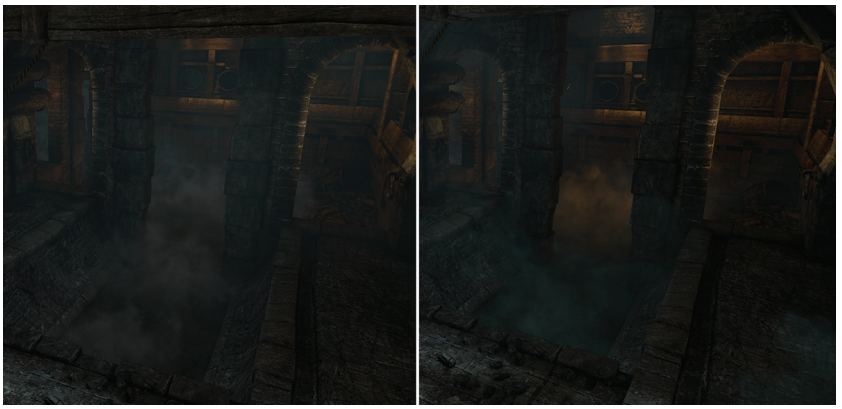

Figure 1.15. Old-generation *Thief* particles rendering (left) and next-generation version (right). Notice the color variation of the fog due to different lighting. [Image courtesy Square Enix Ltd.]

1.4 Lit Particles

Another addition made for the next-generation version of *Thief* was the support for lit particles. Prior to this, our particle system colors had to be tweaked by hand, and this meant that particles could not react to lighting changes. With lighting support, particles look much more integrated into the environment and they appear to react more dynamically to changes in lighting.

The lighting feature set for particles included static light maps, static shadow maps, projected textures, and up to four dynamic lights. We experimented with having dynamic shadow maps, but it revealed too much about the geometry used to render the visual effects (it made it obvious that we were using camera-aligned sprites). Each sprite would have a well-defined shadow mapped as a plane, while the texture used for the particle is usually meant to fake a cloud-like shape. This issue was not visible with static light maps and shadow maps because they were mapped as 3D textures across the particle bounds. Figure 1.15 shows a comparison between old- and next-generation versions of particle rendering in *Thief*.

1.5 Compute-Shader-Based Postprocessing

One of the major novelties of the DirectX 11 API and the next-generation consoles' hardware is the support of compute shaders. One particular feature of compute shaders is the introduction of local data storage (LDS), a.k.a. thread group shared memory (TGSM). LDS is a cache-like on-chip memory, which is generally faster than VRAM but slower than register memory. One can use LDS to exchange data between shader threads running within the same thread group.

Figure 1.16. Gaussian-based DoF with circular bokeh (top) and DoF with hexagonal bokeh (bottom). [Image courtesy Square Enix Ltd.]

This functionality can be used for numerous applications. One obvious use case is decreasing bandwidth for postprocesses, which computes a convolution of a fairly large radius. In *Thief*, we used this feature for depth-of-field (DoF) computations, as will be described below.

1.5.1 Depth-of-Field Rendering

For our DoF algorithm we used two approaches: Gaussian blur for the round-shaped bokeh and [White and Barré-Brisebois 11] for hexagonal bokeh. Figure 1.16 shows examples of these techniques. Both approaches result in two separable filter passes. DoF is texture-fetch limited, as kernels take a big number of samples to accommodate a large radius bokeh.

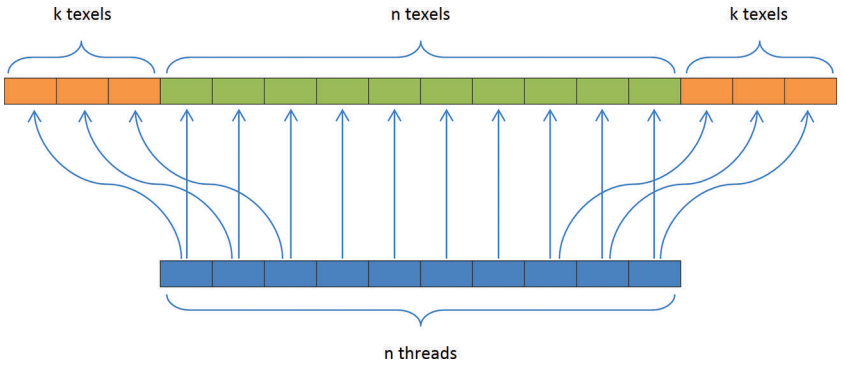

Figure 1.17. Fetching of texels with a filter kernel using local data storage.

1.5.2 Improving Bandwidth by Using Local Data Storage

In order to reduce the texture bandwidth for the DoF pass, we use LDS. The main idea is for a filter of radius k and n threads to prefetch $2k + n$ samples as shown in Figure 1.17. Each of the n threads loads a texel; additionally, every thread close to the thread group boundaries loads another texel. Then, each thread stores the values it loaded into LDS. Finally, to compute the kernel, each thread reads values from LDS instead of DRAM, hence the bandwidth reduction.

Initially, we used the code from Listing 1.6 to load and store from LDS. However, this resulted in even worse performance than not using LDS at all. The reason for this is a four-way LDS memory bank conflict, which we introduced. As bank size on the majority of the video cards is 32-bits wide, each thread will make a strided access with $stride = 4$. To fix that, we needed to de-vectorize our code, as shown in Listing 1.7.

1.5.3 Results

To understand the LDS win, we tested different implementations of the DoF kernel filters. For a DoF pass using a kernel with $radius = 15$ for a FP16 render target, we got 0.15 ms without LDS, 0.26 with vectorized LDS structure, and 0.1 ms for de-vectorized LDS on AMD HD7970. Both next-generation consoles have shown a speedup with a similar factor. In contrast, using LDS on NVIDIA GPUs (GeForce 660 GTX) resulted in no speedup at all in the best case. As a result, on AMD GPUs (which include next-generation consoles), using compute shaders with LDS can result in a significant (33%) speedup if low-level performance considerations (e.g., banked memory) are taken into account.

```
groupshared float4 fCache[NR_THREADS + 2 * KERNEL_RADIUS];

Texture2D inputTexture : register(t0);
RWTexture2D<float4> outputTexture : register(u0);

[numthreads(NR_THREADS, 1, 1)]
void main(uint3 groupThreadID : SV_GroupThreadID,
          uint3 dispatchThreadID : SV_DispatchThreadID)
{
    //Read texture to LDS
    int counter = 0;
    for (int t = groupThreadID.x;
             t < NR_THREADS + 2 * KERNEL_RADIUS;
             t += NR_THREADS, counter += NR_THREADS)
    {
        int x = clamp(
            dispatchThreadID.x + counter - KERNEL_RADIUS,
            0, inputTexture.Length.x - 1);
        fCache[t] = inputTexture[int2(x, dispatchThreadID.y)];
    }
    GroupMemoryBarrierWithGroupSync();

    ...
    //Do the actual blur
    ...

    outputTexture[dispatchThreadID.xy] = vOutColor;
}
```

Listing 1.6. Initial kernel implementation. Notice a single LDS allocation.

1.6 Conclusion

In this chapter, we gave a comprehensive walkthrough for the rendering techniques we implemented for the next-generation versions of *Thief*. We presented our reflection system, the contact-hardening shadow algorithm, particles lighting approach, and compute shader postprocesses. Most of these techniques were integrated during the later stages of *Thief* production, therefore they were used less extensively in the game than we wished. However, we hope that this postmortem will help game developers to start using the techniques, which were not practical on the previous console generation.

1.7 Acknowledgments

We would like to thank Robbert-Jan Brems, David Gallardo, Nicolas Longchamps, Francis Maheux, and the entire *Thief* team.

```
groupshared float fCacheR[NR_THREADS + 2 * KERNEL_RADIUS];
groupshared float fCacheG[NR_THREADS + 2 * KERNEL_RADIUS];
groupshared float fCacheB[NR_THREADS + 2 * KERNEL_RADIUS];
groupshared float fCacheA[NR_THREADS + 2 * KERNEL_RADIUS];

Texture2D inputTexture : register(t0);
RWTexture2D<float4> outputTexture : register(u0);

[numthreads(NR_THREADS, 1, 1)]
void main(uint3 groupThreadID : SV_GroupThreadID,
       uint3 dispatchThreadID : SV_DispatchThreadID)
{
    //Read texture to LDS
    int counter = 0;
    for (int t = groupThreadID.x;
             t < NR_THREADS + 2 * KERNEL_RADIUS;
             t += NR_THREADS, counter += NR_THREADS)
    {
        int x = clamp(
            dispatchThreadID.x + counter - KERNEL_RADIUS,
            0, inputTexture.Length.x - 1);
        float4 tex = inputTexture[int2(x, dispatchThreadID.y)];
        fCacheR[t] = tex.r;
        fCacheG[t] = tex.g;
        fCacheB[t] = tex.b;
        fCacheA[t] = tex.a;
    }
    GroupMemoryBarrierWithGroupSync();

    ...
    //Do the actual blur
    ...

    outputTexture[dispatchThreadID.xy] = vOutColor;
}
```

Listing 1.7. Final kernel implementation. Notice that we make a separate LDS allocation for each channel.

Bibliography

[Andersson 13] Zap Andersson. "Everything You Always Wanted to Know About mia_material." Presented in Physically Based Shading in Theory and Practice, SIGGRAPH Course, Anaheim, CA, July 21–25, 2013.

[Andreev 13] Dmitry Andreev. "Rendering Tricks in Dead Space 3." Game Developers Conference course, San Francisco, CA, March 25–29, 2013.

[Fernando 05] Randima Fernando. "Percentage-Closer Soft Shadows." In *ACM SIGGRAPH 2005 Sketches*, p. article 35. New York: ACM, 2005.

[Gruen and Story 09] Holger Gruen and Jon Story. "Taking Advantage of Direct3D 10.1 Features to Accelerate Performance and Enhance Quality." Pre-

sented at AMD sponsored session, Eurographics, Munich, Germany, March 30–April 3, 2009.

[Gruen 10] Holger Gruen. "Contact Hardening Shadows 11." AMD Radeon SDK, http://developer.amd.com/tools-and-sdks/graphics-development/amd-radeon-sdk/archive/, 2010.

[Kasyan et al. 11] Nick Kasyan, Nicolas Schulz, and Tiago Sousa. "Secrets of CryENGINE 3 Graphics Technology." SIGGRAPH course, Vancouver, Canada, August 8, 2011.

[Lagarde and Zanuttini 13] Sébastien Lagarde and Antoine Zanuttini. "Practical Planar Reflections Using Cubemaps and Image Proxies." In *GPU Pro 4: Advanced Rendering Techniques*, edited by Wolfgang Engel, pp. 51–68. Boca Raton, FL: CRC Press, 2013.

[Lengyel 11] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics*, Third edition. Boston: Cengage Learning PTR, 2011.

[NVIDIA Corporation 99] NVIDIA Corporation. "Cube Map OpenGL Tutorial." http://www.nvidia.com/object/cube_map_ogl_tutorial.html, 1999.

[Sousa 05] Tiago Sousa. "Generic Refraction Simulation." In *GPU Gems 2*, edited by Matt Farr, pp. 295–305. Reading, MA: Addison-Wesley Professional, 2005.

[Uludag 14] Yasin Uludag. "Hi-Z Screen-Space Cone-Traced Reflections." In *GPU Pro 5: Advanced Rendering Techniques*, edited by Wolfgang Engel, pp. 149–192. Boca Raton, FL: CRC Press, 2014.

[White and Barré-Brisebois 11] John White and Colin Barré-Brisebois. "More Performance! Five Rendering Ideas from *Battlefield 3* and *Need For Speed: The Run*." Presented in Advances in Real-Time Rendering in Games, SIGGRAPH Course, Vancouver, August 7–11, 2011.

[Wright 11] Daniel Wright. "Image Based Reflections." http://udn.epicgames.com/Three/ImageBasedReflections.html, 2011.