Screen Space Cone Tracing for Glossy Reflections

Bachelorthesis by Lukas Hermanns
email: lukas.hermanns@igd.fraunhofer.de

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Date: March 24, 2015
Supervisor: Priv.-Doz. Dr. Arjan Kuijper
Supervisor: M.Sc. (Inf.) Tobias A. Franke

Thesis Statement pursuant to §22 paragraph 7 of APB TU Darmstadt

I, Lukas Hermanns, herewith formally declare that I have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form. In the submitted thesis the written copies and the electronic version are identical in content.

Date: Signature:

Abstract

Indirect lighting (also Global Illumination (GI)) is an important part of photo-realistic imagery and has become a widely used method in real-time graphics applications, such as Computer Aided Design (CAD), Augmented Realtiy (AR) and video games. Path tracing can already achieve photorealism by shooting thousands or millions of rays into a 3D scene for every pixel, which results in computational overhead exceeding real-time budgets. However, with modern programmable shader pipelines, a fusion of ray-casting algorithms and rasterization is possible, i.e. methods, which are similar to testing rays against geometry, can be performed on the GPU within a fragment (or rather pixel-) shader. Nevertheless, many implementations for real-time GI still trace perfect specular reflections only.

In this Bachelor thesis the advantages and disadvantages of different reflection methods are exposed and a combination of some of these is presented, which circumvents artifacts in the rendering and provides a stable, temporally coherent image enhancement. The benefits and failings of this new method are clearly separated as well. Moreover the developed algorithm can be implemented as pure post-process, which can easily be integrated into an existing rendering pipeline. The core idea of this thesis has been presented as a poster at SIGGRAPH 2014 [Hermanns and Franke, 2014].

Contents

1 Introduction

1.1 Overview

In this Bachelor thesis a novel method for real-time glossy reflections is presented. This method can be implemented as a pure post-process, which simplifies the effort, of integrating it into an existing graphics application, considerably. Similar algorithms are already implemented, not only in video games, but also in CAD and AR applications and reflections for indirect lighting are essential in GI simulations. These reflections increase the photorealism of Computer Generated Imagery (CGI) drastically (see Figure 1.1).

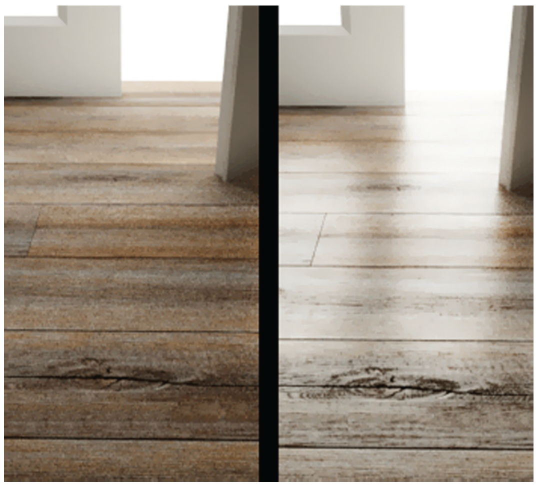


Figure 1.1: Photorealism is increased by reflections on the right, while the picture on the left lacks this feature. Image courtesy of *GPU Pro 5*.

1.2 Motivation

Indirect lighting is implemented in a broad range of GI applications since many years, particularly in film productions such as *Terminator 2* (1991), *Avatar* (2009), and the first feature-length computer animated motion picture *Toy Story* (1995) [Henne and Hickel, 1996]. From the very beginning, developers are striving to reduce the calculation overhead or rather to accelerate the rendering process. This is because accurate rendering can take several hours or days even on high performance computer systems, depending on the scene and shading complexity. Therefore the costs and duration to procude photo-realsic imagery can be very high.

To solve this, there are many algorithms to achieve photorealism by approximating the influence of indirect light, for both real-time and non real-time rendering. Some of these simulate the indirect light by distributing many small direct light emitters throughout the scene, called Virtual Point Light (VPL), but which then requires new optimizations or rather hierarchies of spatial data structures. Still others store and propagate the indirect light through the scene within a volume, called Light Propagation Volume (LPV), which then causes a large memory footprint for large scenes and high resolution volumes. However, methods like VPL and LPV lack an important part of indirect light, which will be discussed in this thesis and is shown in the above image.

Our novel approach for real-time reflections is a rough approximation but still produces plausible effects of indirect lighting. It is fast, fully dynamic, and can easily be integrated into an existing rendering pipeline, because it is a pure post-process. Hence, the result of an intermediate render pass is the only input for this effect. No information about the scene is required. Nevertheless, even this approach is not a final solution for all cases of GI application. This is also discussed in this thesis and summarized in the conclusion.

1.3 Outline

Here is a brief outline of the following chapters:

Chapter 2: Fundamentals
: Gathers the fundamental terms, required for real-time reflections in computer graphics. Declares, what *direct-* and *indirect lighting* is and declares the terms *diffuse*, *specular*, and *glossy* in the context of the *microfacet model*. Also, a basic understanding of *ray tracing* and *cone tracing* is conveyed. Supplementary, a transition from *ray tracing* to its screen space counterpart is presented, and respectively for *cone tracing*.

Chapter 3: Related Work
: Presents seven examples of related work: one for direct and six for indirect lighting. Several examples have been taken from current, but also from older, video games such as *Quake III Arena* (1999), *Remember Me* (2013), and *Killzone Shadow Fall* (2013). Also, a state of the art method for real-time glossy reflections, called *Hi-Z Screen-Space Cone-Traced Reflections* from the book *GPU Pro 5*, is presented, which constitutes the base of our novel algorithm. Finally, a comparison between these examples is made.

Chapter 4: Screen Space Cone Tracing
: Explains the noval method in detail, with formulas, pseudocode, shader examples, and illustrations. This chapter starts with a simple implementation for screen space reflections, which is later used as fallback for special cases in the actual algorithm. Subsequently, the core algorithm is divided into five sections. Finally, the chapter also examines the implementation of a competitor called SSLR and gives some optimization advices.

Chapter 5: Results and Discussion
: Shows various performance and image quality comparisons between our novel method, its base algorithm, one of its competitors, and the ground truth. Two different scenes are used for image comparison: a custom-made model which shows two small rooms with low scene complexity and the well-known *Sponza Atrium* model which shows a large atrium with high scene complexity. The system setup, which has been used to render the results, is also specified in detail. Finally, a table of performance benchmarks with several screen resolutions and material roughness parameters is presented.

Chapter 6: Conclusion
: Summarizes the benefits and failings of the novel method compared to the related work. Prospective ideas for further enhancements are proposed.

2 Fundamentals

Before we can look at *state of the art* methods for real-time reflections, we need to consider some fundamentals. We need to know what direct and indirect lighting actually means, and we need to know how it can be computed. We then cross over from ray tracing to approximations in screen space.

2.1 Diffuse, Specular, and Glossy Reflections

We start out with the terms *diffuse*, *specular*, and *glossy* reflections. With direct lighting the illumination for each pixel is computed by calculating the energy of all surrounding light emitters the pixel reflects. On opaque surfaces it is commonly computed with a Bidirectional Reflectance Distribution Function (BRDF). In the simplest case a plain scalar product between the surface normal $\vec{N}$ and the light vector $\vec{L}$ is computed to obtain the light intensity I (see Equation 2.1) which is then multiplied with the energy of the light source.

$$I_{direct} = \max\left\{0, \vec{N} \cdot \vec{L}\right\} \tag{2.1}$$

The max function clamps the scalar product to the interval $[0,1]$ to avoid negative values. For uniformity all vectors in a shading setup are meant to point away from the shaded point (see Figure 2.1).

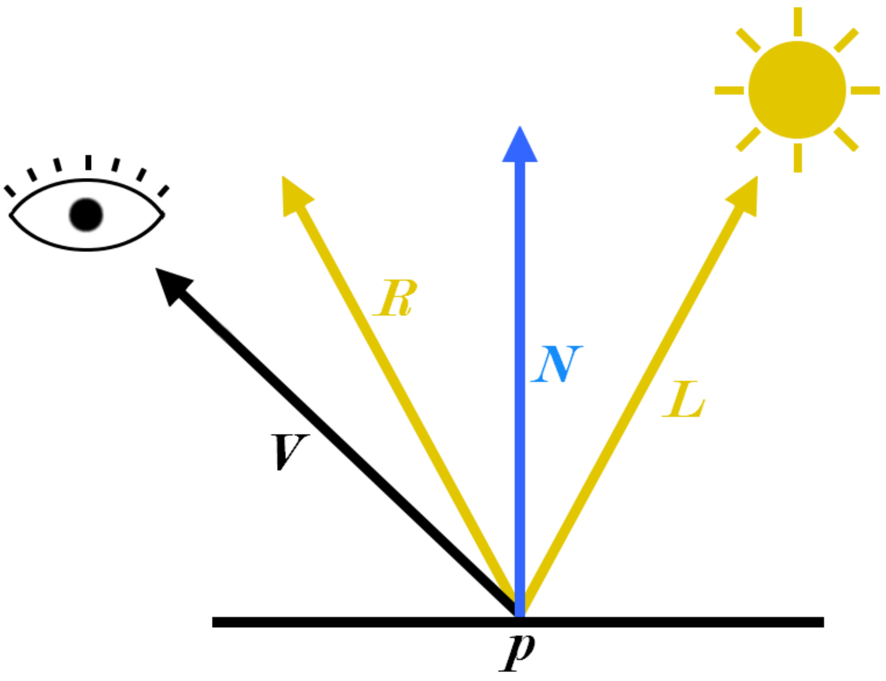


Figure 2.1: Basic shading setup for point p with view vector $\vec{V}$, surface normal $\vec{N}$, light vector $\vec{L}$, and reflection vector $\vec{R}$ (see Equation 2.2).

$$\vec{R}_{reflect} = \vec{N} \times 2 \times \left(\vec{N} \cdot \vec{L}\right) - \vec{L} \tag{2.2}$$

Equation 2.1, also known as Lambert's cosine law, is a simple way to model diffuse reflection. It is a rough approximation to a sub-surface scattering phenomenon when light interacts with diffuse surfaces. Sub-surface scattering happens when light rays impact materials and leave them at different positions and with different directions. This is due to the nature of light at the atomic level. But even modern hardware does not provide enough computational power to calculate the

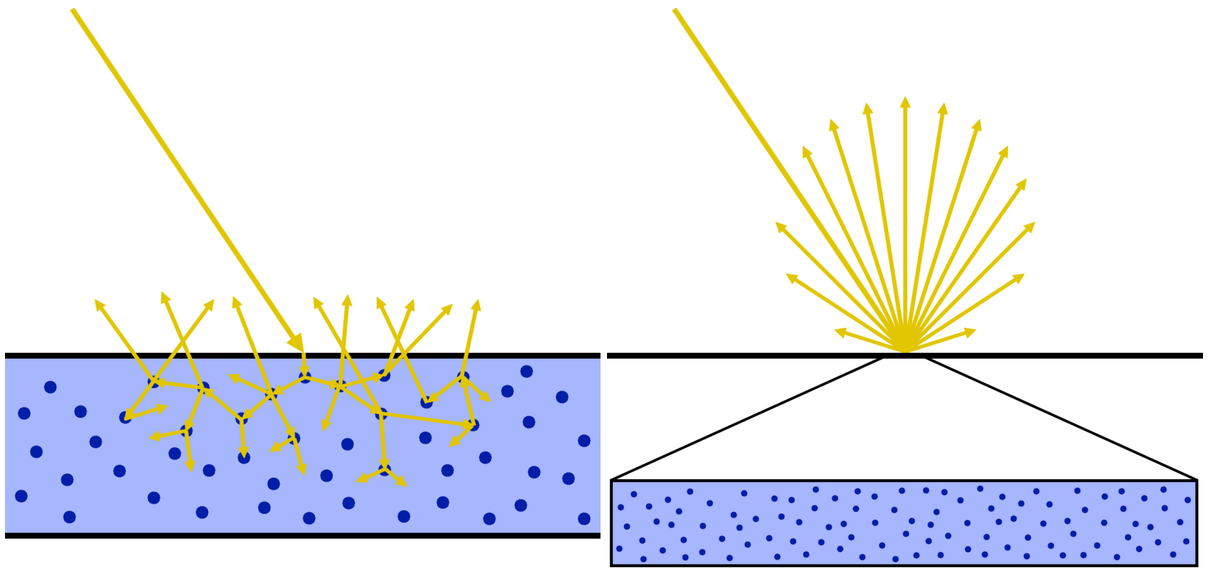


Figure 2.2: With light scattering the rays interact with the matter, but with diffuse lighting the rays are simulated by simply reflecting into all directions.

light rays interacting with each atom. Hence, we simplify this with a model within a simulation — whereat a simulation is always a simplification of the reality. With diffuse lighting we approximate the scattering by ignoring the displacement and reflecting the light rays into all directions within the hemisphere over the point of impact (see Figure 2.2). However, the diffuse term is only one part of realistic lighting. The second part is distinguished by *specular* and *glossy* reflections:

- *Specular*: Reflections on perfect mirrors ($\varrho = 0$).

- *Glossy*: Reflections on rough surfaces ($\varrho > 0$). Deviates the light rays from the reflection ray.

These reflections are based on the *microfacet model* where each surface is assumed to consist of very tiny perfect mirrors — the microfacets. The surface roughness ϱ specifies how much the normal vectors of these microfacets deviate from each other. When the microfacets are arranged identically we get perfect specular reflections and when the microfacets are arranged irregular we get glossy reflections (see Figure 2.3). This deviation can be computed with a *Cook-Torrance* BRDF [Cook and Torrance, 1982].

The final illumination of a pixel is then computed by the diffuse and specular terms, which is commonly just *Diffuse* + *Specular*.

2.2 Indirect Lighting

So far we only considered direct lighting, i.e. computing the light which is directly emitted from a light source, reflected from a surface and viewed by the camera; no further reflection paths are traced. For this reason we will now focus on indirect lighting (also GI).

The *rendering equation* [Kajiya, 1986] describes how *radiance* leaving a point is given as the sum of emitted plus reflected radiance. With direct lighting we only integrate the energy from light emitters but not from reflectors. And in fact everything reflects light. In ray tracing — which will be discussed in the next section — one possibility is to sample the hemisphere around the shaded point to receive all incident light. See Figure 2.4 for a comparison between direct and indirect lighting.

In summary: indirect lighting is the process of accumulating all the light to a point along the lights reflection paths. Every time we trace a reflection path we talk about a *bounce*. The most important bounce of indirect lighting is the first one due to light attenuation (see Figure 2.5). The computational complexity of indirect lighting is straightforward for perfect mirrors but becomes very extensive for rough surfaces, because the number of ray samples may increase exponentially. Moreover, many real-time GI applications implement only the first bounce as rough approximation.

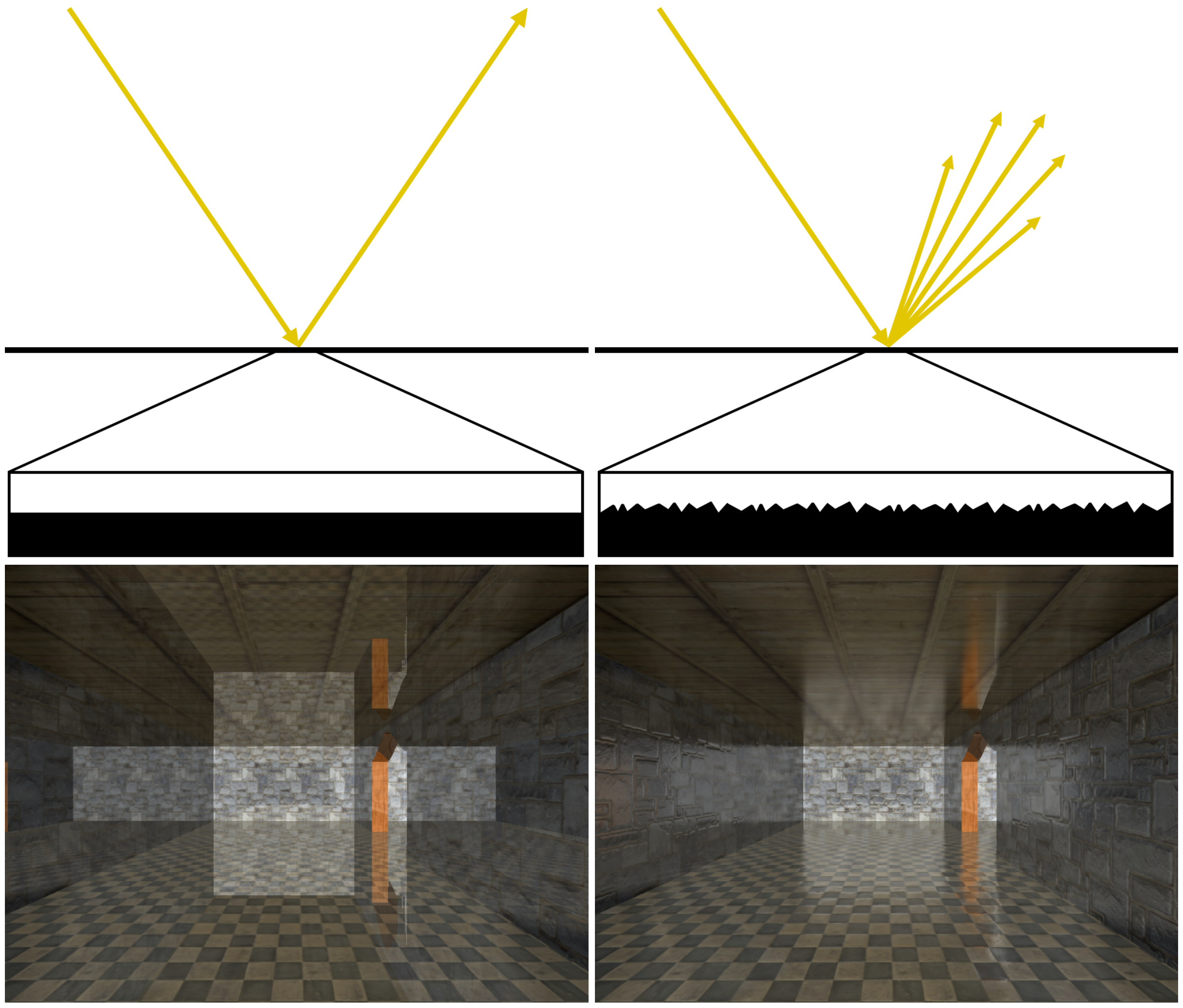

(a) Specular (b) Glossy

Figure 2.3: Difference between perfect specular- and glossy reflections. The microfacets are illustrated as tiny perfect reflectors in the upper right image. The viewer looks along a corridor in a shiny environment.

2.3 Ray Tracing

Optics is *rays*, *waves*, and *photons*. There are many ways to simulate the behavior of light but we will focus on the *geometrical optics approximation*, which can be done by tracing rays. We already mentioned the term *ray tracing* which is a subtopic of *ray casting*. Ray casting is the process of shooting rays into a 3D scene and doing tests against its geometries, e.g. to find an intersection between a ray and a triangle, whereas ray tracing is the process of shooting rays from the eye into a 3D scene and trace their reflection paths. For further topics, such as path tracing, we defer the interessted reader to [Dutre et al., 2006].

To perform ray tracing we need the geometry data of the entire scene. In *rasterization* — which is done on common GPUs — the geometry is rendered by plotting each geometric primitive (such as triangles) instead of sampling the scene by ray casts. Although modern GPUs provide powerful shader units, which allows us to construct and traverse complex spatial data structures and hierarchies, ray tracing is still a time-consuming process. For each single ray cast a tree hierarchy (such as *Octree*, *kd-Tree*, *Bounding Volume Hierarchy (BVH)* or *Binary Space Partitioning (BSP)*) must be traversed and intersection tests of 'ray against triangle' must be performed for polygonal meshes. This depends on the scene complexity and the quality of hierarchy construction. Moreover hierarchy traversal usually causes incoherent memory access, which is a further performance hit.

Therefore, accurate ray tracing is still done mainly on the CPU (or rather many CPUs) for *offline rendering* (opposite of real-time rendering), where the synthesis of a single picture may take several hours. Some example ray tracing frameworks are: *RenderMan* [Upstill, 1989] (by PIXAR), *POV-Ray* [POV-Team, 2004] (by THE POV-TEAM) and *Embree* [Wald et al., 2014] (by INTEL). A basic algorithm for ray tracing (specular reflections only) is shown in Algorithm 1.

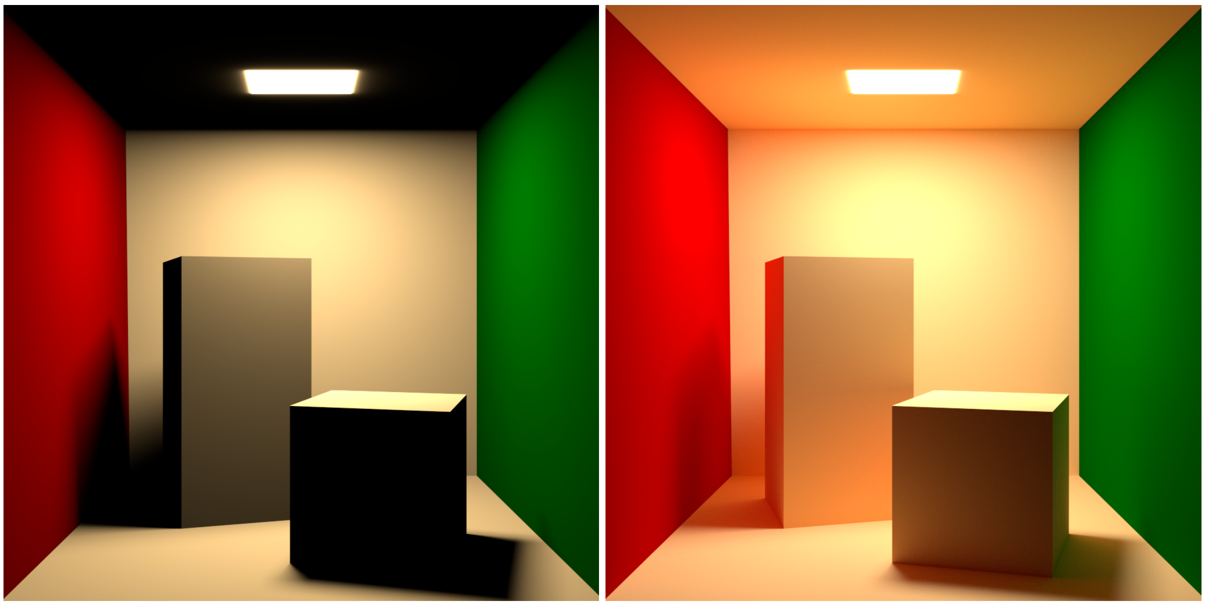

(a) Direct lighting **(b)** Direct + indirect lighting

Figure 2.4: Comparison between direct and indirect lighting illustrated by the *Cornell Box* [Goral et al., 1984] scene. Rendered with *Path Tracing* in *Blender* [Blender-Foundation, 1994].

2.4 Ray Tracing in Screen Space

This thesis is about *screen space cone tracing* and as already mentioned ray tracing is actually too expensive for real-time rendering. But we can approximate this process even further, by tracing rays only in screen space. Afterwards, we will proceed on to cone tracing.

In summary, ray tracing in screen space means, that we trace a ray only with the information we have from the screen, which are at least the colors and depth values for each pixel. It turns out, that lots of information get lost by this rough approximation. Though, it can enhance the photo-realistic appearance notably. In addition we don't distinguish between ray casting and ray tracing since the ray casts are only used for shading, and not for general collision detection. Furthermore, note that with 'screen space' we mean a set of *textures* with the size of the screen resolution. This set of textures is typically named Geometry Buffer (G-Buffer). This is because with a G-Buffer we can reconstruct the geometry which is visible inside the viewport, if it contains a *depth texture*. 'Reconstruction' in this context means that the pixel coordinate is transformed to its global position in world- or view space. Besides the pixel depth we also need the inverse projection matrix to *unproject* the screen coordinate into a world (or view) coordinate. With the following formulas we can transform a coordinate from screen space (P_{SS}) into view space (P_{VS}), whereat M_{proj}^{-1} specifies the inverse projection matrix:

$$\begin{aligned} P_{SS_{xy}} &:= \left(P_{SS_{xy}} - (0.5, 0.5)^T\right) \times (2, -2)^T \\ P_{PS} &:= M_{proj}^{-1} \times P_{SS} \\ P_{VS} &:= P_{PS_{xyz}} / P_{PS_w} \end{aligned} \tag{2.3}$$

In a common post-processing pipeline a G-Buffer with *color*-, *normal*- and *depth* textures is available. The color texture contains the colors of the previous render pass. The normal texture contains the normal vectors — encoded as colors (see Transformation 2.4) — to compute the reflection vector for each individual pixel.

$$f(x) = \begin{cases} [-1,1] & \rightarrow [0,1], \\ x & \mapsto \frac{x+1}{2}. \end{cases}, f^{-1}(x) = \begin{cases} [0,1] & \rightarrow [-1,1], \\ x & \mapsto x \times 2 - 1. \end{cases} \tag{2.4}$$

And the depth texture contains the pixel depths which is required to reconstruct the position (in *world*- or *view* space) for each pixel. In a G-Buffer the normal vectors from the normal buffer are commonly in view space, thus the reflection ray is in view space, too. But to trace a ray in screen space we also need the reflection ray to be in screen space. To do this we first compute the reflection ray in view space and then project it back into screen space. This works quite similar to the formulas 2.3 and is shown in more detail in Section 4.2.2.

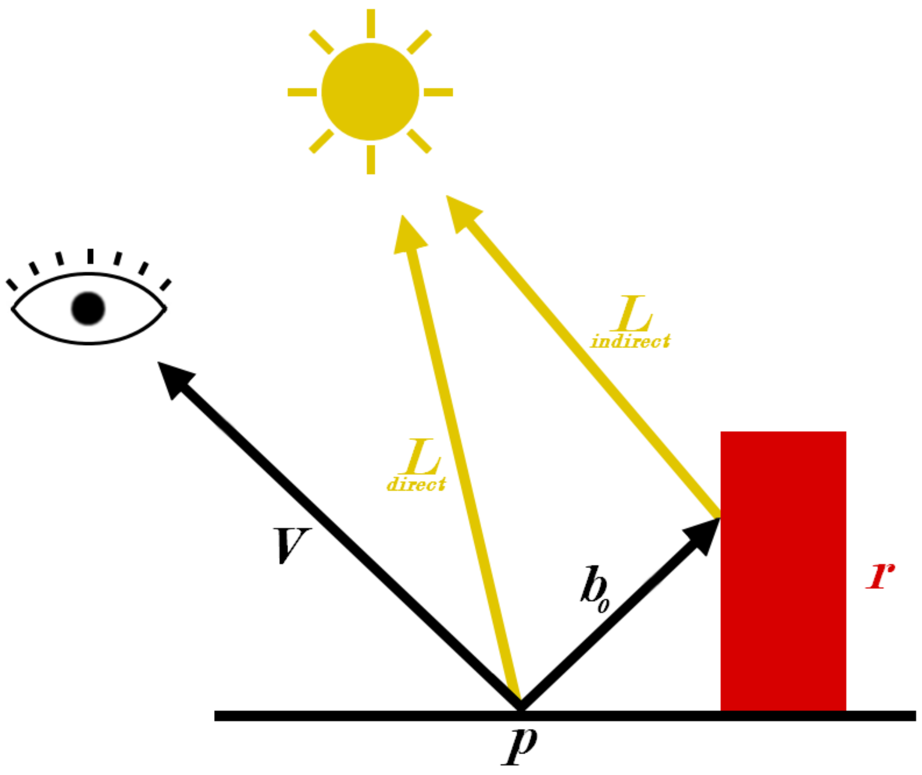


Figure 2.5: Direct lighting L_{direct}, indirect lighting $L_{indirect}$, and first bounce b_0 to reflector r.

Once we have all these information we can move along the reflection ray and check if we step through the depth texture. This happens if the Z value of the current 'ray position' is greater than or equal to the depth value at that texture coordinate (see Inequation 2.5).

$$Z_{ray} \geq Z_{gbuffer} \tag{2.5}$$

The first question that arises here is: how long is our step size to move along the ray? If the step size is too long we may skip the correct geometry intersection. If the step size is too short the process may take too long. A proper solution to this is *dynamic adaptation*: we start with a small step size, increase it in every iteration and do a *binary search* if we found the first intersection. The implementation for this is explained in more detail in section 4.

In addition to precision, we have two more problems: *screen space limits* and *hidden geometry*. The former originates from the fact that we can only reconstruct the geometry which is visible inside the viewport. We don't have the geometry data of the entire scene. The latter originates from the fact that we project a 3D scene into a 2D G-Buffer (see Transformation 2.6).

$$project : \mathbb{R}^3 \rightarrow \mathbb{N}^2 \tag{2.6}$$

If geometry is covered by another geometry, we can not reconstruct it as well (see Figure 2.6). There are several workarounds to these limitations we will also discuss in section 4.

2.5 Cone Tracing

In the original publication of *cone tracing* (*Ray tracing with cones* [Amanatides, 1984]), the definition of a "ray" has been extended into a cone by including information on the spread angle (see Figure 2.7). When glossy or diffuse reflections are simulated with ray tracing, many rays must be cast recursively for each bounce. To decrease the number of ray samples drastically the idea of *cone tracing* has been expanded. A modern technique with this approach is Voxel Cone Tracing (VCT) [Crassin et al., 2011]. The core idea of VCT is to pre-filter all direct lighting in the scene and then sample at a specific filter level. Such a pre-filter is typically stored inside the Multum In Parvo (MIP)-maps (Latin 'a multitude in a small space') of a texture. In VCT a 3D texture or an octree structure is used to store this information. The cone itself is further approximated by several texture lookups which produces something like a set of slices of a pyramid. Nevertheless, the method is called cone tracing to denote the *cone aperture*.

Algorithm 1 Ray Tracing Basic Algorithm where $f_{shininess} \in [0,1]$ specifies the shininess factor to accumulate reflection color. This example allows specular reflections only.

```
 1: procedure REFLECT($\vec{I}, \vec{N}$)                                  ▷ Incident vector $\vec{I}$ and normal $\vec{N}$
 2:     return $\vec{I} - \vec{N} \times 2 \times (\vec{N} \cdot \vec{I})$                ▷ Small adjustment of Equation 2.2
 3: procedure RAYCAST($R, S$)                                          ▷ Ray and scene hierarchy
 4:     $I \leftarrow \infty$                                                      ▷ Initialize intersection
 5:     $D_{prev} \leftarrow \infty$                                               ▷ Initialize previous distance
 6:     $N_{list} \leftarrow$ FINDLEAFNODES($S_{root}, R_{origin}, R_{direction}$)
 7:     for all $L \in N_{list}$ do                                        ▷ Iterate over leaf node list
 8:         for all $T \in L_{triangles}$ do                               ▷ Iterate over triangle list
 9:             $P \leftarrow$ FINDINTERSECTION($R, T$)                      ▷ Find intersection point P
10:             $D_{new} \leftarrow$ DISTANCE($R_{origin}, P$)                    ▷ Get new distance
11:             if $D_{new} < D_{prev}$ then
12:                 $D_{prev} \leftarrow D_{new}$                                  ▷ Store nearest distance
13:                 $I_{point} \leftarrow P$                                  ▷ Store nearest intersection point
14:                 $I_{normal} \leftarrow T_{normal}$                             ▷ Store normal from nearest point
15:     return $I$                                                     ▷ Return nearest intersection
16: procedure RAYTRACE($R, S, L, N_{iteration}$)                         ▷ Lights L and iteration count
17:     $I \leftarrow$ RAYCAST($R, S$)                                       ▷ Find nearest intersection point
18:     $C_{pixel} \leftarrow$ SHADEPOINT($I, L$)                               ▷ Shade pixel color with lights L
19:     if $N_{iteration} > 0$ then
20:         $M_{origin} \leftarrow I_{point}$
21:         $M_{direction} \leftarrow$ REFLECT($R_{direction}, I_{normal}$)                ▷ Get reflection ray M
22:         $C_{pixel} \leftarrow C_{pixel} +$ RAYTRACE($M, S, L, N_{iteration} - 1$) $\times f_{shininess}$
23:     return $C_{pixel}$                                                 ▷ Return shaded pixel color
```

2.6 Cone Tracing in Screen Space

The idea of VCT is adopted into screen space by using the MIP-maps of a 2D texture instead of a 3D texture. In this case the cone is abstracted by an isosceles triangle, to which we will refer as a cone because of the conceptual similarities. There are several approaches to trace a cone in screen space. Those that have been used in Screen Space Cone Tracing (SSCT) [Hermanns and Franke, 2014] and HZCT [Uludag, 2014] are very different. While in SSCT we sample from a specific MIP level in each iteration, in HZCT the ray tracing and cone tracing parts are clearly separated.

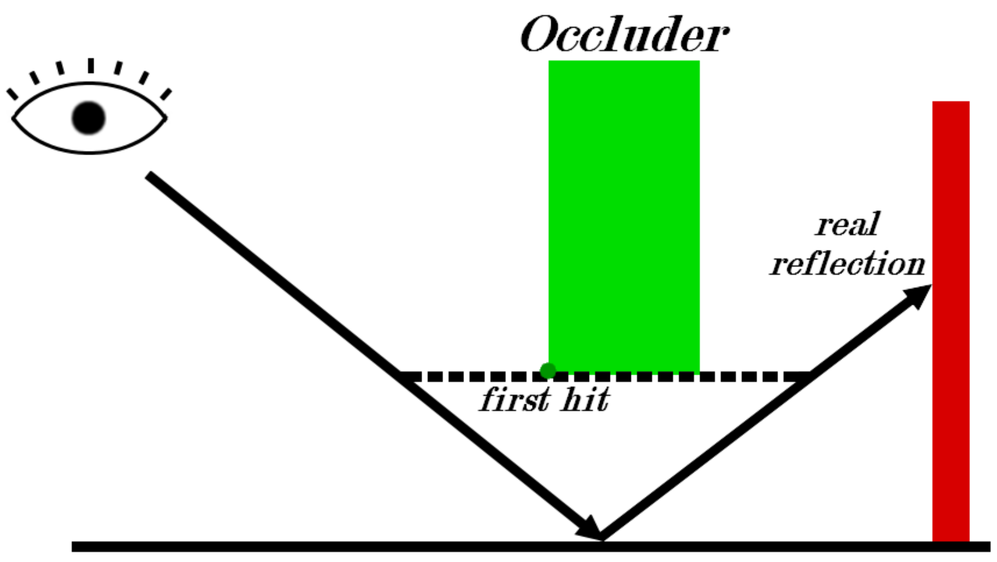


Figure 2.6: Hidden geometry can not be reconstructed in screen space.

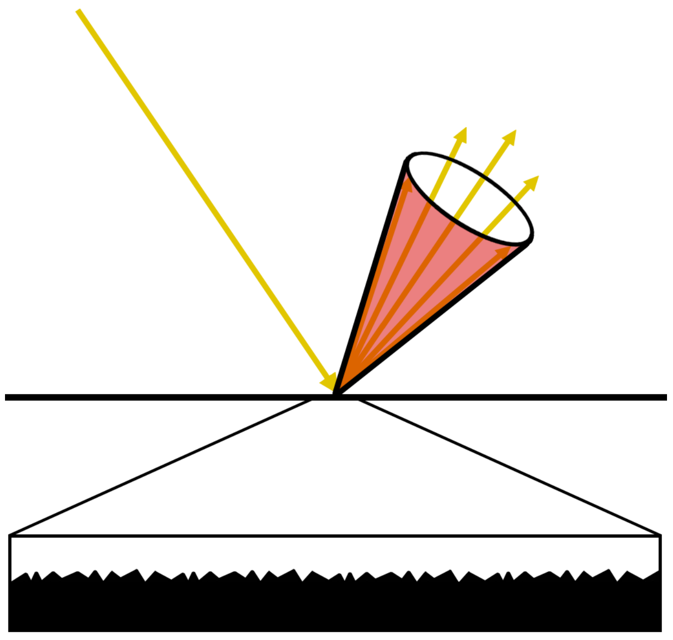


Figure 2.7: Cone aperture (in red) represents the approximation of several rays.

3 Related Work

In this chapter we will analyze several approaches for real-time reflections and then make a comparison to summarize their benefits and failings.

3.1 Approaches

3.1.1 Analytical Area Lights

We start out with a simple method for glossy reflections which deals with direct lighting only: *Analytical Area Light (AAL)*.

The three basic models of light shapes are: *Directional*-, *Point*-, and *Spot* light. In classical fixed function pipelines these are the usual lighting primitives to shade geometry. All of them only have a position and optionally a direction, but neither an area nor volume. This makes it very easy and fast to determine the light intensity for each pixel, because only a scalar product and the distance between that pixel and the respective light source must be computed. However, every light source in reality has a volume. This can be simulated more plausible with AALs, which cover a specific *area* on the screen. The formulas for these light models resemble the basics of ray tracing. Ray intersection tests are performed against all AALs for every pixel, otherwise the nearest distance between the current pixel and the primitive light geometry is computed. Since these light models require more complex and more flexible calculations, they became popular with the advances of programmable shader units in modern hardware, but the research around them has begun long before [Campbell and Fussell, 1991]. In addition, glossy reflections can be simulated easily by adding a further light attenuation depending on the surface roughness.

Contrariwise, they only allow direct lighting, since with indirect lighting everything is treated as a light source, but AALs only provide limited shapes, such as cuboids, spheres, cylinders, and capsules (see Figure 3.1). Although there are more complex AALs, such as fractals, these shapes are not enough to represent an entire reflectable scene. Therefore, one can use them only as direct light emitters.

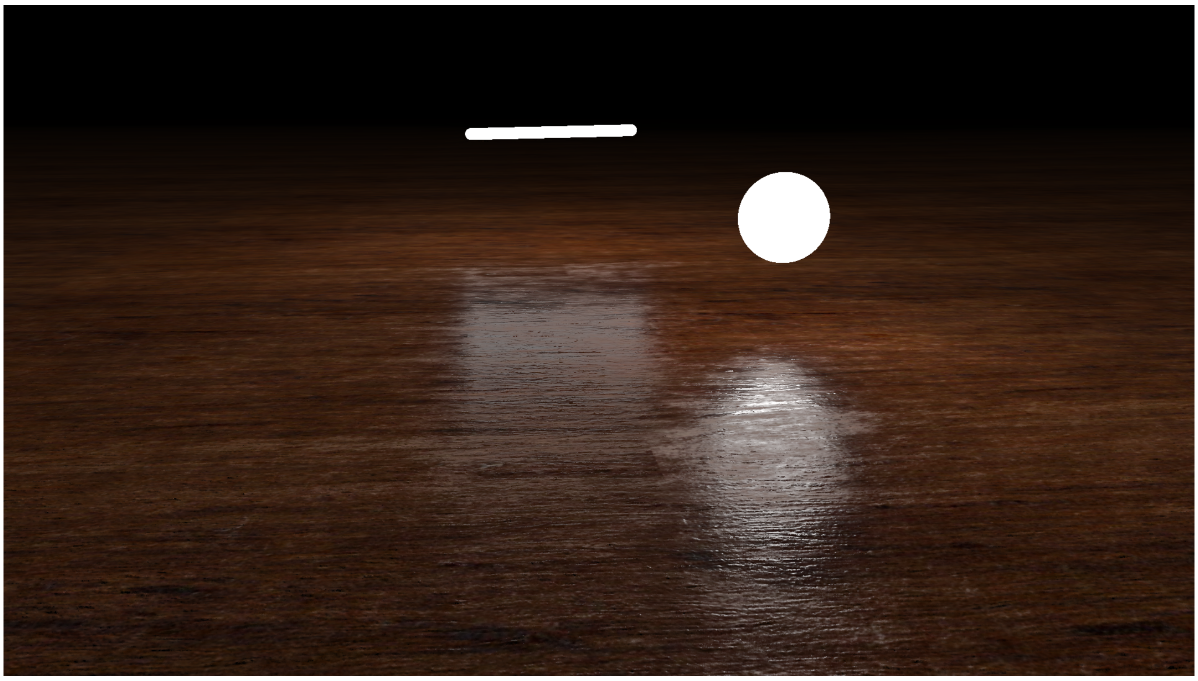


Figure 3.1: Reflections of two AALs with direct lighting [Shadertoy, 2013]

3.1.2 Planar Reflections

We continue with one of the oldest methods for reflections of indirect lighting, which is still used in modern 3D engines: *Planar Reflections*.

For planar reflections there are two major methods: the first one is to use a *stencil buffer* and the second one is to use a *render target* (also *framebuffer*). With the first method, the stencil buffer operates — as the name implies — like a stencil. It allows to reject pixels outside the stencil pattern. The reflected scene should only be visible inside this pattern, and to generate it, the reflective geometry is rendered with certain render states enabled. We can divided this algorithm into the following steps:

1. Clear frame- and stencil buffers.
2. Render scene with default settings but without reflective geometry.
3. Render reflective geometry into stencil buffer.
4. Render scene inside the stencil pattern with mirrored view transformation.

In the second method, the steps of rendering the actual and the mirrored scene are in the reverse order. We first have to render the mirrored scene into the render target. Then the render target serves as the source of the texture (and also of the reflective light color), which is mapped onto the reflective geometry, when the actual scene is rendered. In some cases this method may be used but it has a larger memory footprint.

This is a very simple method which allows perfect specular reflections with correct geometry reconstruction. The correct reconstruction originates from the fact that the scene is rendered a second time, which can be very time consuming. This also means that the performance depends on the scene complexity. Another disadvantage is that this only works for planar reflectors such as *mirrors* or flat *water puddles*, because the mirroring is due to a mirrored matrix transformation, which does not allow any distortions. Note that this method has nothing to do with ray tracing or something similar. It is just a secondary rasterization pass. However, there are also hybrid approaches which combine geometry- and image-based rendering, such as *Forward Mapped Planar Mirror Reflections* [Bastos and Stürzlinger, 1998].

Such a technique is used in the *idTech 3* game engine from 1999 (see Figure 3.2) and even older ones.

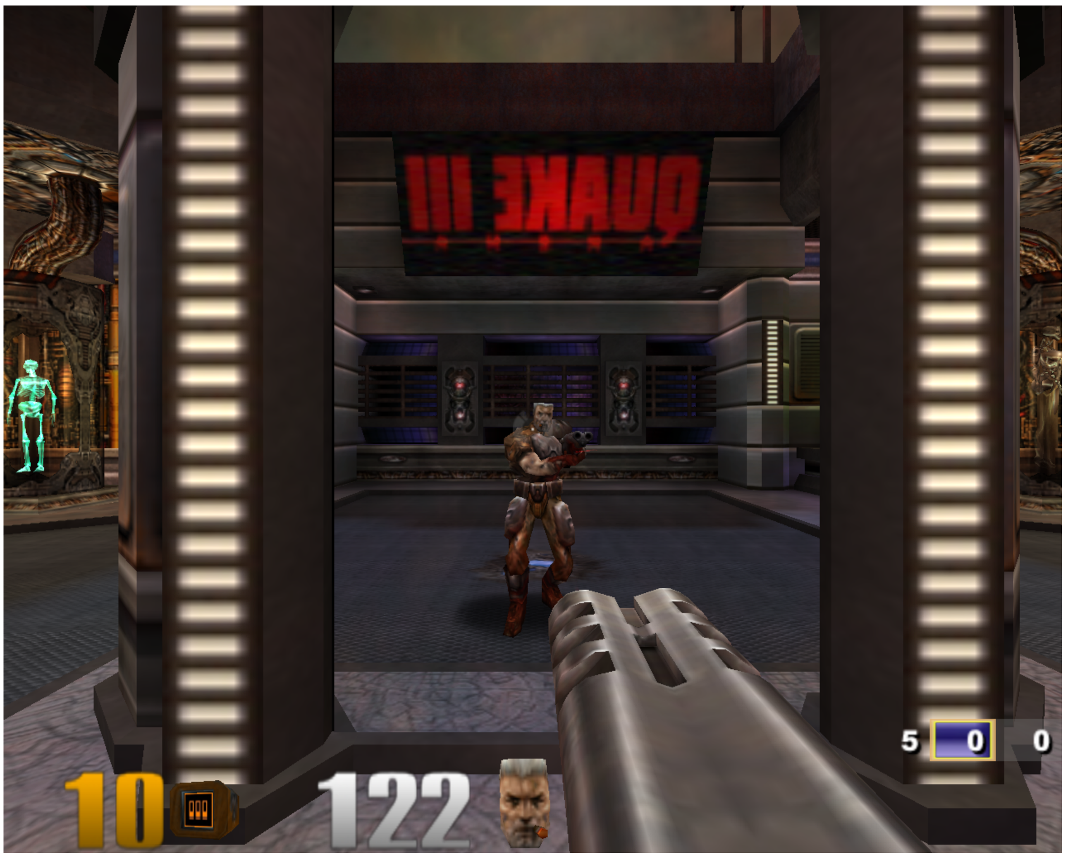


Figure 3.2: Mirror in *Quake III Arena* reflecting the Player and Quake Logo.

3.1.3 Environment Maps

Another still widely used method are (localized-) *Environment Maps* (also *Cube Maps*).

In this case the scene is rendered from six view angles into a cube map at a localized position. A cube map internally consists of six texture layers, one for each cube face. This captures a 360° view in a single texture. These cube maps are typically placed by an artist within a world editor and generated during the world building process — this process is also called *baking*. Such a cube map stores the entire light influence for the point where the cube map was rendered. Usually the cube maps are additionally pre-filtered (or blurred) to efficiently simulate a basic glossiness for all surfaces [Kautz and McCool, 2000]. The blurring simulates a wider distribution of reflection rays, which causes a glossy appearance. As already mentioned: the reflections from each cube map are only correct for a single 3D point. Whenever a cube map is used for other points, the reflections are only correct for infinitely distant light. Such a candidate is the (nearly) infinitely distant sun.

To overcome this restriction several cube maps are generated in a scene, between which the application must choose at runtime. To avoid the *popping effect* (discrete switching between textures), some applications interpolate between the N nearest cube maps (the popping effect is visible in games like *Half-Life 2* and *Portal* using the *Source Engine*). A further improvement are *parallax corrected cube maps* [Lagarde and Antoine, 2012], which can adjust the singular location to a cubic environment. However, even this method is severely limited due to its cubic nature and needs further adaptation by artists.

This technique is used in the game *Remember Me* from 2013 (see Figure 3.3) and the *Source Engine* from 2004.

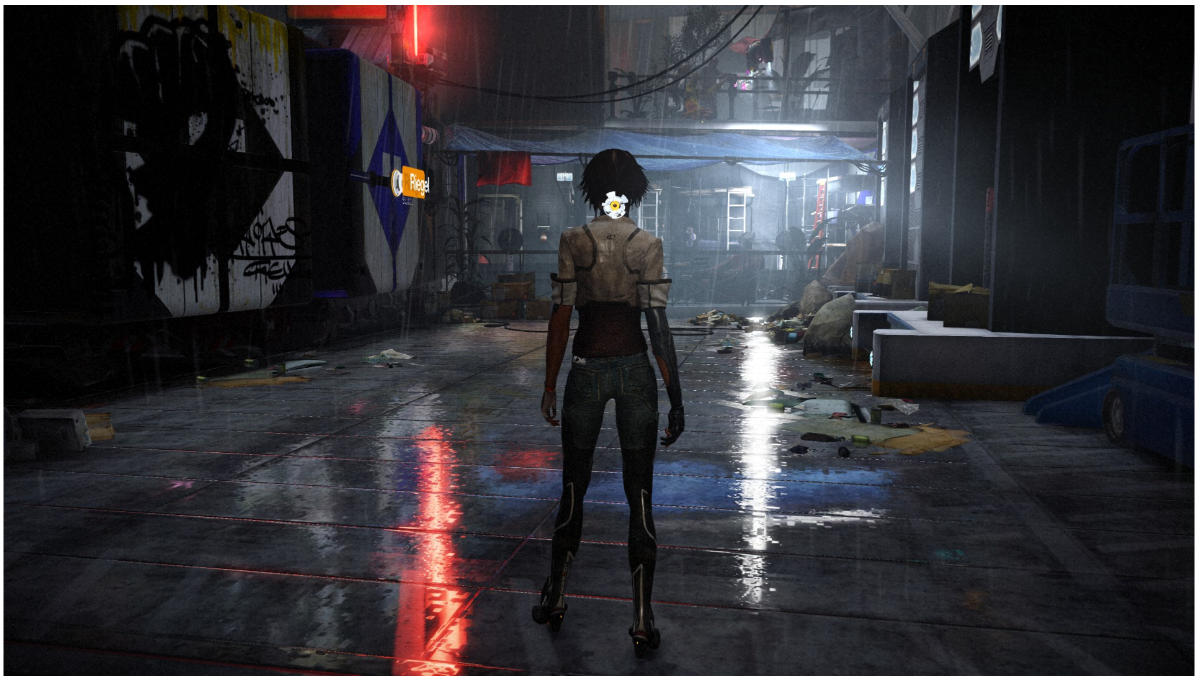


Figure 3.3: Specular highlights are visible on the lower left, rendered with cube maps in *Remember Me* from 2013.

3.1.4 Image Based Reflections

We now continue with ‘ray tracing like’ methods: *Image Based Reflections (IBR)*.

In this method several IBR proxies are placed by an artist in the world editor, whereat an IBR proxy is a box which captures a small volume in the scene which is then rendered into a 2D texture. For each reflection ray an intersection test against the plane, which is spanned by that box, is computed. These ‘ray against plane’ tests are very fast but this reflection model is only useful for ‘nearly’ planar reflectors such as building facades or streets for instance. Although a single plane intersection test is fast some form of hierarchy is required if many IBR proxies are used (e.g. 50+) to avoid testing against all planes for every pixel. In addition the method is inappropriate for perfect specular reflections since the geometry inside an IBR box is approximated by a plane. For glossy reflections the results are reasonable because the distorted planar reflector can not be perceived exactly by the viewer.

This technique is used in the game *Thief 4* [Sikachev and Longchamps, 2014] from 2014 and the *Unreal Engine 3 (UE3)* [Wright] from 2006 (see Figure 3.4).

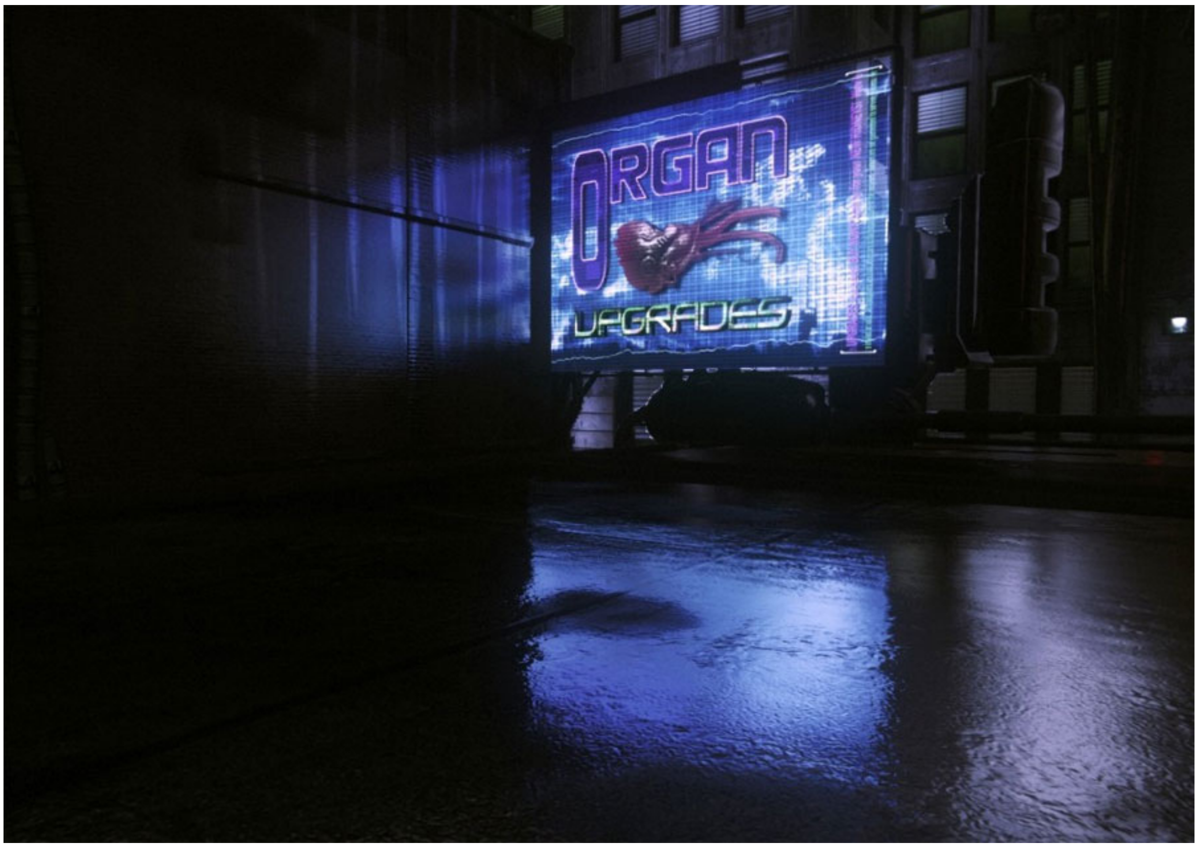


Figure 3.4: IBRs are visible at the floor and walls in this *UE3* demo (Copyright ©2001-2012 Epic Games, Inc).

3.1.5 Screen Space Local Reflections

Finally we look at a pixel based ray tracing method: *Screen Space Local Reflections (SSLR)*.

In a nutshell: we transform the reflection ray from view space into screen space, and then move along this ray until we 'step through the depth buffer'. By this algorithm, we hope to find the intersection of a ray against the scene geometry, which is stored in form of the depth buffer. That means, in particular, we can only find intersections with geometry, which is already visible on the screen. This is why it is called a *screen space* effect. Many SSLR implementations perform this simple *ray hit search* only, which is commonly called a *linear ray march*. Once the first 'hit' is found, a binary search for refinement can be done. To simulate glossy reflections most applications apply subsequent blur passes to the reflection color buffer. But there are also alternatives where several rays are casted and the average color forms the result. A frequently used optimization, to find the ray intersections, is to render this pass only at half resolution $\left(\frac{Width}{2}, \frac{Height}{2}\right)$, which yields to acceptable results when a blur pass is applied anyway.

The advantage of SSLR is that it allows reflections of arbitrary geometry (assumed the geometry is visible on screen). It can additionally be implemented as a pure post-process or a set of consecutive post-processes, which alleviates the effort to integrate it into a present 3D engine. It is moreover independent of the scene complexity because the reflections are fetched from the color buffer of a previous render pass. The calculations are computed for every pixel which makes the algorithm's effort proportional to the number of pixels on the screen. Disadvantages are primarily the limitations of the screen boundary and the hidden geometry problem.

This technique is used in *UE3* and the game *Killzone Shadow Fall* [Valient, 2014] from 2013 (see Figure 3.5) which will later be discussed in more detail. Such ray tracing approximations are state of the art in the field of post-processing for which several scientific publications exist (see [Johnsson, 2012] and [McGuire and Mara, 2014]).

3.1.6 Screen Space Cone Tracing

Based on local reflections in screen space, we move on to a method which traces cones instead of rays: *SSCT*.

The basics of this method has already been presented in section 2.6. Again, the 'cone' is an approximation for many reflection rays. The process is very similar to SSLR but in each iteration of the ray march we sample from a certain MIP-map of the depth texture, which is a further approximation of the actual 'cone' (see Figure 3.6). By relying on MIP-maps

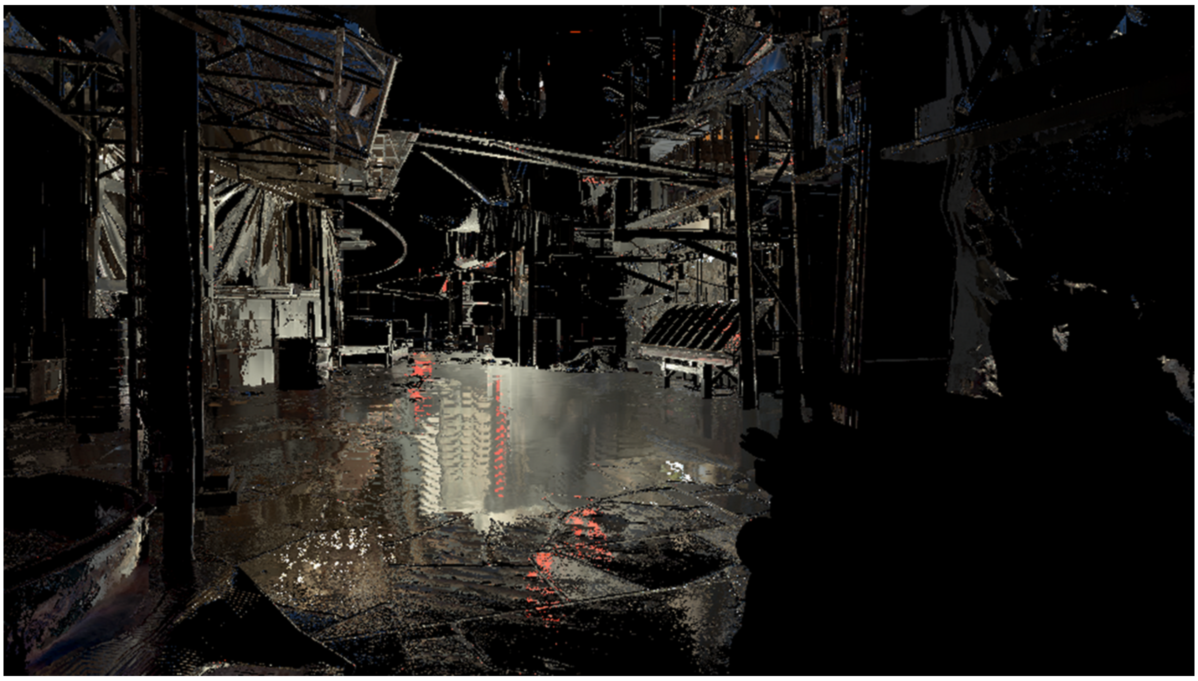


Figure 3.5: 'Raytrace color buffer' of an intermediate render pass in *Killzone Shadow Fall*.

certain integration errors are unavoidable. This is due to solid angles that can subtend either flat spaces or multiple pieces of geometry. It can manifest as alias or temporal inconsistency when moving the camera view. Such errors will increase notably as the cone angle size increases to simulate more glossy appearances. Thus for SSCT using a proper texture filter is crucial. Next to naïve MIP-mapping, manual filtering by using slices of a 3D texture, where each slice is a gaussian blurred version of the original texture, has also been tested. The results are significantly better in quality but loose the fast texture accesses since we sample from a high resolution 3D texture.

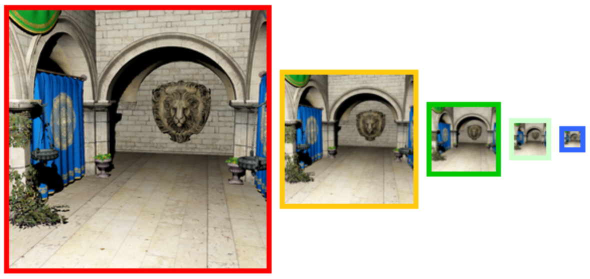
This technique is shown in the SSCT poster at SIGGRAPH'14 [Hermanns and Franke, 2014] (see Figure 3.7).

3.1.7 Hi-Z Cone Tracing

The last presented method is from the book *GPU Pro 5*: *HZCT* [Uludag, 2014].

The remarkable concept with this method is on the one hand that the ray tracing and cone tracing parts are clearly separated and on the other hand that the ray tracing process is accelerated with hierarchical buffers (Hi-Z buffer [Hi-Z $\triangleq$ Hierarchical Z] and visibility buffer). These hierarchical buffers will be generated during the post-process in each frame and allow a faster, stable, and precise ray tracing in screen space. While SSLR and SSCT use a binary search to refine the intersection point, HZCT is much more target-oriented. However, a disadvantage is that HZCT does not allow tracing rays which point towards the camera. This is due to the ray setup in combination with the hierarchical buffers. As soon as an intersection has been detected, the cone tracing pass integrates all incident radiance from the intersection point to the ray origin using the visibility buffer. The cone approximation is quite similar to that in SSCT but it is combined with a visibility buffer to circumvent invalid integration over a large solid cone angle. The actual algorithm will be explained in more detail in section 4.

For an example scene, rendered with HZCT, see Figure 3.8.


(a) MIP maps

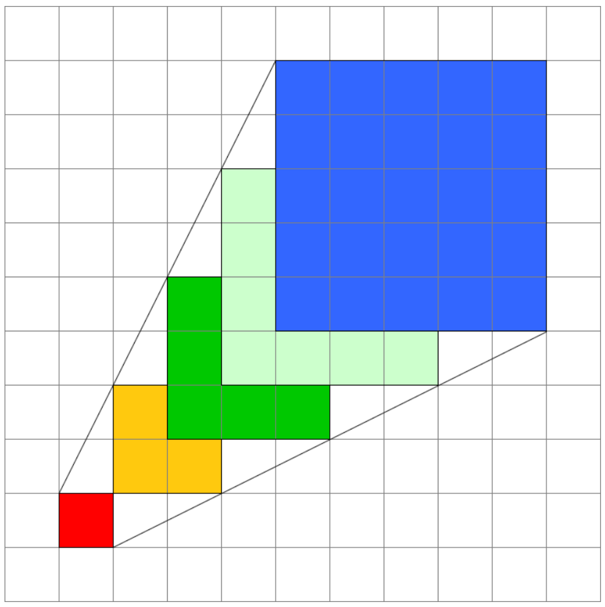
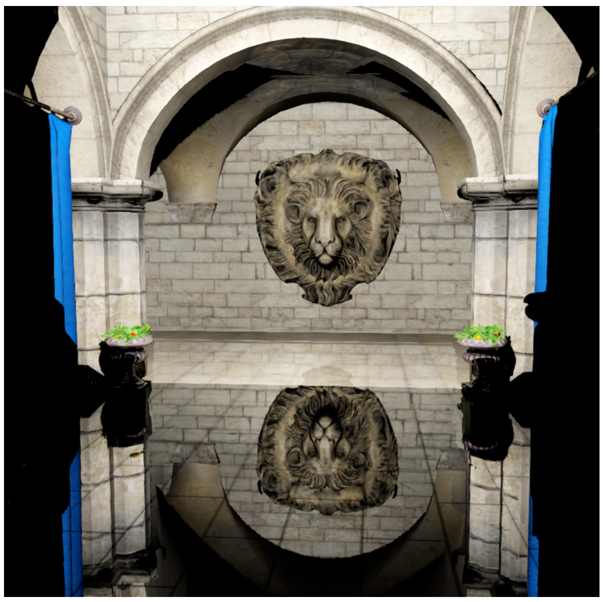

(b) Cone approximation

Figure 3.6: Illustrates the rough approximation of the cone. Each quad represents a single pixel sample of a pre-filtered image. At any point on the cone its distance and solid angle can be used to determine a filter level.


(a) Specular reflections

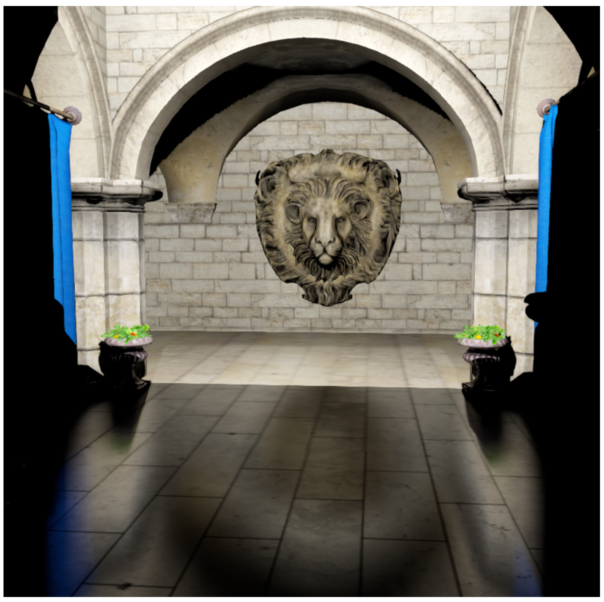

(b) Glossy reflections

Figure 3.7: Comparison between specular reflections with no roughness (3.7a) and glossy reflections with high roughness (3.7b) in the *Sponza Atrium* [Dabrovic, 2002].

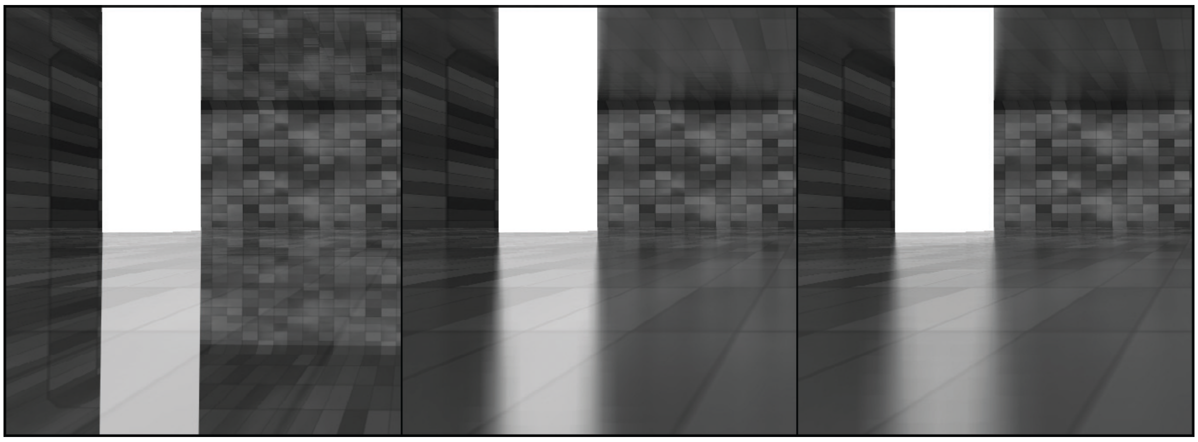


Figure 3.8: Varying roughness rendered with *Hi-Z Cone Tracing*. Image courtesy of *GPU Pro 5*.

3.2 Comparison

We have seen some related work and state of the art techniques, so we can now make a comparison between these approaches:

Pros	Cons
Analytical Area Lights	
• Accurate calculation for incident radiance • Easy to implement	• Direct lighting only
Planar Reflections	
• Perfect reconstruction of reflected geometry • Easy to implement	• Depends on scene complexity • Planar reflectors only
Environment Maps	
• Easy to implement • Very fast	• Must be placed by artist • Limited to fixed count • Only correct for infinitely distant light
Image Based Reflections	
• Good visual approximation • Very fast	• Must be placed by artist • 'Nearly' planar reflectors only
Screen Space Local Reflections	
• Reflection of arbitrary geometry • Pure post process	• Hiden geometry problem • Limited to screen space
Screen Space Cone Tracing	
• Reflection of arbitrary geometry • Pure post process • Glossy reflections with arbitrary roughness	• Hiden geometry problem • Limited to screen space • Artifact avoidance is very slow
Hi-Z Cone Tracing	
• Reflection of arbitrary geometry • Pure post process • Glossy reflections with arbitrary roughness • High stability and precision • Acceleration with hierarchical buffers	• Hiden geometry problem • Limited to screen space • Unable to ray trace towards the camera • Complex to implement

The only method which provides perfect reconstruction of reflected geometry here are *planar reflections*. All the other techniques merely approximate the reflections in a more or less coarse manner. Unfortunately, planar reflections are inappropriate for every reflective geometry which has not a planar shape. Moreover the necessity of re-rendering the scene makes it unfeasible for post-processing. Nevertheless, planar reflection is still a valuable fallback method, especially when others fail with receiving scene information.

A very efficient method here are *environment maps*, which is at least beneficial as fallback, too. Primarily because they can be pre-computed. But either with or without parallax correction, using environment maps requires some adaptation by artists, i.e. the reflections can not be computed as a pure post-process. At least the localization within the scene must be managed with a world editor. This also applies to IBR.

From the screen space ray- and cone tracing methods, HZCT seems to be the most advanced technique. The major benefits over SSLR and SSCT are the stability, the precision, and the acceleration. However, like most ray tracing algorithms in screen space, it is limited to the screen boundary and we have the hidden geometry problem. Even HZCT does not solve these restrictions, but this is where fallback methods are considered.

4 Screen Space Cone Tracing

Finally we start with the actual algorithm for cone tracing in screen space. The core idea of this thesis is based on the mentioned SIGGRAPH'14 SSCT poster. However, the implementation is based on HZCT and will be slightly augmented with a fallback for rays pointing towards the camera, to maximize the extent of reflection rays in screen space. We will then provide a quality comparison to the original HZCT and a method from the field of SSLR.

4.1 Overview

The previous chapter presented benefits and failings of related work. The most advanced technique for screen space reflections seems to be HZCT. However, it does not cover the maximum extent of the screen space, because rays pointing towards the camera can not be gathered. Fortunately, this method separates ray tracing and cone tracing, thus the ray tracing part can be augmented with *linear ray marching* for exceptional cases. After we review the algorithm in detail we will take a closer look at a couple of SSLR methods. One of them is a technique used in *Killzone Shadow Fall*, in which a *mask buffer* is generated alongside the ray tracing process. This is later used to enhance the blurring of the ray trace color buffer, which is required for glossy reflections in this SSLR method. The blurred variants are stored inside the MIP-chain.

4.2 Implementation

The code samples in this chapter are either in pseudocode or the OpenGL Shading Language (GLSL) [Khronos-Group, 2004].

4.2.1 Depth Buffer

First of all, we need to define what kind of depth buffer we use during the entire algorithm. There are two particular kinds of depth buffers: a *linear*- and a *post-projected* depth buffer. The GPU uses a post-projected depth buffer per default, whereas the depth (also 'Z'-) values are in the range $[0, 1]$ and the values near to the camera (i.e. near to 0) have higher precision than the values far away (i.e. near to 1). This is an instrumental distribution of floating-point precision, because objects near to the camera must be more precise than objects far away. However, some shaders use a linear depth buffer, in which the Z values are either in the range $[0, 1]$ or just the distance between the camera and the pixel position in world space, but both are linearly distributed. Shadow mapping for instance can be implemented much easier with linear depth buffers. But screen space effects have to use a post-projected depth buffer when they interpolate their depth values. This is due to the projection, where linear values can not be interpolated linearly in a correct manner. Though post-projected values can be linearly interpolated properly in screen space [Low, 2002].

We can exploit this fact in our ray tracing process. Therefore we use a post-projected depth buffer throughout the entire algorithm.

4.2.2 Linear Ray March

To get started with ray tracing in screen space, recall the simple and frequently used *linear ray march*:

```
#define LINEAR_MARCH_COUNT  100
#define INVALID_RAY         vec3(-1.0)

vec3 LinearRayMarch(vec3 ray, vec3 dir, float stride) {
    for (int i = 0; i < LINEAR_MARCH_COUNT; ++i) {
        // Sample depth value at current ray position
        float depth = textureLod(depthBuffer, ray.xy, 0.0);

        // Check if ray steps through the depth buffer
        if (ray.z > depth)
            return ray; // Return final intersection

        // Step to the next sample position
        ray += dir * stride;
    }
    return INVALID_RAY; // No intersection found
}
```

Note that line number 14, where *ray* is increased by *dir* × *stride*, is only valid because we use a post-projected depth buffer. In a linear depth buffer this interpolation produces incorrect results when comparing ray_z with *depth*.

The above code is a very primitive example but as already mentioned *dynamic adaptation* can be used to accelerate the ray marching and a *binary search* can be used to refine the final intersection point:

```
 1  #define LINEAR_MARCH_COUNT      100
 2  #define LINEAR_MARCH_INCREASE   1.5
 3  #define BINARY_SEARCH_COUNT     32
 4  #define BINARY_SEARCH_DECREASE  0.5
 5  #define DEPTH_DELTA_EPSILON     0.01
 6  #define INVALID_RAY             vec3(-1.0)
 7
 8  vec3 LinearRayMarch(vec3 ray, vec3 dir, float stride) {
 9      vec3 prevRay = ray;
10
11      for (int i = 0; i < LINEAR_MARCH_COUNT; ++i) {
12          // Sample depth value at current ray position
13          float depth = textureLod(depthBuffer, ray.xy, 0.0);
14
15          // Check if ray steps through the depth buffer
16          if (ray.z > depth) {
17              // Intersection found -> now do a binary search
18              return BinarySearch((prevRay+ray) * 0.5, dir * stride);
19          }
20
21          // Store previous position and step to the next sample position
22          prevRay = ray;
23          ray += dir * stride;
24
25          // Increase stride to accelerate ray marching
26          stride *= LINEAR_MARCH_INCREASE;
27      }
28
29      return INVALID_RAY; // No intersection found
30  }
31
32  vec3 BinarySearch(vec3 ray, vec3 dir) {
33      for (int i = 0; i < BINARY_SEARCH_COUNT; ++i) {
34          // Sample depth value at current ray position
35          float depth = textureLod(depthBuffer, ray.xy, 0.0);
36
37          // Check if the 'depth delta' is smaller than our epsilon
38          float depthDelta = depth - ray.z;
39          if (abs(depthDelta) < DEPTH_DELTA_EPSILON)
40              break; // Final intersection found -> break iteration
41
42          // Move ray forwards if we are in front of geometry,
43          // and move ray backwards if we are behind geometry.
44          if (depthDelta > 0.0)
45              ray += dir;
46          else
47              ray -= dir;
48
49          // Decrease direction vector for further refinement
50          dir *= BINARY_SEARCH_DECREASE;
51      }
52
53      return ray; // Intersection already found, but we could not refine it to the minimum
54  }
```

This is an example implementation as well, but in practical graphics applications there may be much more conditional branches for corner cases. The recent code sample increases the step size (or rather *stride*) in each iteration by a factor of 1.5 (see line number 26). In practice, we also check if the ray left the screen boundary, in which case we set the ray back to its previous position and reduce the step size (This is omitted in the code sample to keep it simple). We call the BinarySearch procedure as soon as the first 'hit' is found. From now on the binary search tries to refine the intersection point. The iteration counts, the increase/decrease factors, and the Δ_ϵ are all 'heuristic values' which must be determined by trial and error, i.e. there is no prove that these values are the best solution, but should work for most use cases.

We will use this procedure afterwards for the rays which point towards the camera, i.e. those that can not be gathered by HZCT.

Now we have a ray tracing procedure which works in screen space. But so far we have not yet discussed how to determine the reflection ray in screen space. This is implemented in the following algorithm:

Algorithm 2 Determine **Reflection Ray** in Screen Space.

```
1:  procedure PROJECT($P_{VS}, M_{proj}$)    ▷ Point in view space $P_{VS}$ and projection matrix $M_{proj}$
2:      $P_{PS} \leftarrow M_{proj} \times P_{VS}$    ▷ Transform $P_{VS}$ into projection space ($P_{PS}$)
3:      $P_{SS} \leftarrow P_{PS_{xyz}} / P_{PS_w}$    ▷ Unproject point with division by $W$
4:      $P_{SS_{xy}} \leftarrow P_{SS_{xy}} \times (0.5, -0.5)^T + (0.5, 0.5)^T$    ▷ Transform $X$ and $Y$ from the interval $[-1,1]$ to $[0,1]$ and invert $Y$
5:      return $P_{SS}$    ▷ Return final point in screen space
6:  procedure UNPROJECT($P_{SS}, M_{proj}^{-1}$)    ▷ Point in screen space $P_{SS}$ and inverse projection matrix $M_{proj}^{-1}$
7:      $P_{SS_{xy}} \leftarrow \left(P_{SS_{xy}} - (0.5, 0.5)^T\right) \times (2, -2)^T$    ▷ Transform $X$ and $Y$ from the interval $[0,1]$ to $[-1,1]$ and invert $Y$
8:      $P_{PS} \leftarrow M_{proj}^{-1} \times P_{SS}$    ▷ Transform $P_{SS}$ into projection space ($P_{PS}$)
9:      return $P_{PS_{xyz}} / P_{PS_w}$    ▷ Return unprojected point with division by $W$
10: procedure REFLECTIONRAY($TexCoord, z, N_{VS}, M_{proj}$)
11:     $P_{SS} \leftarrow \left(TexCoord_{xy}, z\right)^T$    ▷ Get pixel position in screen space
12:     $P_{VS} \leftarrow$ UNPROJECT($P_{SS}, M_{proj}^{-1}$)    ▷ Unproject pixel position into view space
13:     $D_{VS} \leftarrow$ NORMALIZE($P_{VS}$)    ▷ Get view direction in view space
14:     $R_{VS} \leftarrow$ REFLECT($D_{VS}, N_{VS}$)    ▷ Get reflection in view space
15:     $O_{SS} \leftarrow$ PROJECT($P_{VS} + R_{VS} \times \epsilon, M_{proj}$)    ▷ Project reflection offset into screen space
16:     $V_{SS} \leftarrow O_{SS} - P_{SS}$    ▷ Get reflection vector in screen space
17:     return $V_{SS}$    ▷ Return final reflection vector
```

The procedure REFLECTIONRAY provides the final direction of our reflection ray V_{SS} in screen space. The input parameters for this procedure are: the coordinate *TexCoord* of the current pixel, the pixel depth z, the normal vector N_{VS} in view space, and the projection matrix M_{proj}.

4.2.3 Overview of HZCT

Since the method is based on HZCT we continue with an overview of the different passes as proposed in *GPU Pro 5*. The core algorithm can be divided into five steps:

Hi-Z Pass
Generates the entire MIP chain of the Hi-Z buffer. This hierarchical buffer is required to accelerate the ray tracing.

Pre-Integration Pass
Generates the entire MIP chain of visibility buffer. This hierarchical buffer is required for accurate radiance integration during cone tracing.

Pre-Convolution Pass
Pre-filters the color buffer. A blur pass of the default MIP-map generation can be used here.

Ray-Tracing Pass
Finds the ray intersections. This pass requires the Hi-Z buffer.

Cone-Tracing Pass
Integrates incident radiance for a solid cone angle. This pass requires the Hi-Z-, the color-, and the visibility buffers.

The last two steps can be combined into a single shader. It is also possible to split them up, but this should only be taken into account when it really matters, because particularly state changes to the framebuffer binding are very time consuming and it would also increase the memory footprint.

4.2.4 Hi-Z Pass

As shown in the previous code samples and commonly implemented in SSLR we accelerate the ray tracing by increasing the step size in each iteration. Steps which are too large may skip over small geometry, which can lead to incorrect intersections. With a *Hierarchical-Z* buffer we can find our intersection fast but also precise. The idea of the Hi-Z buffer is to take the *minimum* or the *maximum* Z value from the original depth buffer and store it in the next lower buffer at half size. In our case we will use the minimum (see Figure 4.1). This is done for each MIP-map. Note that the original

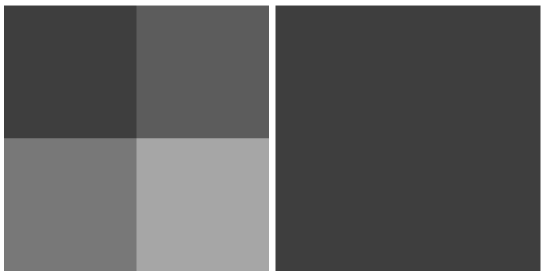

(a) Before **(b)** After

Figure 4.1: Illustration of the Hi-Z buffer generation with the minimum Z value.

buffer is always stored in the first MIP-map (level 0), i.e. the higher the MIP-level the smaller the buffer. This hierarchy allows us to avoid omitting geometries during ray tracing. More about that in section 4.2.7.

Before we continue consider the following pitfall: The default MIP-maps are generated by the GPU. This is a highly optimized process which always takes the average value of the previous MIP-level. Since we need the minimum value, we generate the MIP-maps manually. In most cases it is enough for MIP-map generation to take only four values from the previous MIP-level (as shown in Figure 4.1). However, this is not correct for None Power Of Two (NPOT) resolutions (e.g. 800×600 or 1920×1080). This is due to halving odd widths and heights, where some pixel values get lost. Consider a texture height of 600: $600 \xrightarrow{\div 2} 300 \xrightarrow{\div 2} 150 \xrightarrow{\div 2} 75$. As you can see no matter what size we start with, at some point division-by-two results to odd sizes if we don't have power-of-two resolutions. Since we generate our Z buffer in a hierarchical fashion, this failure exacerbates in each iteration. Hence, we need to adjust the determination of the minimal Z value slightly.

The following pseudocode shows the main part for generating each MIP-map. To reduce (not completely prevent, but reduce) the failure, desribed above, we adapt the right and bottom texture lookups with "offset".

Algorithm 3 Hi-Z Buffer Generation.

```
1: procedure HiZPixelShader($Tex, P_{pixel}, Offset$)          ▷ Input parameters for the Hi-Z pixel shader
2:     $P_{sample} \leftarrow P_{pixel} \times 2$                  ▷ Sample position is twice the pixel coordinate
3:     $z_0 \leftarrow \text{ReadDepth}(Tex, P_{sample} + (0, 0)^T)$
4:     $z_1 \leftarrow \text{ReadDepth}(Tex, P_{sample} + (Offset_x, 0)^T)$
5:     $z_2 \leftarrow \text{ReadDepth}(Tex, P_{sample} + (0, Offset_y)^T)$
6:     $z_3 \leftarrow \text{ReadDepth}(Tex, P_{sample} + (Offset_x, Offset_y)^T)$
7:     $z_{min} \leftarrow \min\{z_0, z_1, z_2, z_3\}$             ▷ Write $z_{min}$ to the pixel output
```

Definition 1 (texel) *A texel here denotes the pixel of a texture (abbr. for "**Tex**ture-**El**ement").*
The values of $Offset_x$ and $Offset_y$ are either 1 or 2 (see Equation 4.1).

$$Offset_x = \begin{cases} 1 & \text{if } Width \text{ is even,} \\ 2 & \text{else.} \end{cases}, Offset_y = \begin{cases} 1 & \text{if } Height \text{ is even,} \\ 2 & \text{else.} \end{cases} \tag{4.1}$$

As the interested reader may already have realized we still only fetch four values in all cases. We only adapt the right and bottom texture lookups. This approach leads to good performance and acceptable quality results. In other words, we don't generate a 100% accurate Hi-Z buffer, but we capture the approximated minimal Z value for the texel boundaries in the MIP chain (See Figure 4.2 and Figure 4.3).

Now we have our Hi-Z buffer and we can continue with the next passes.

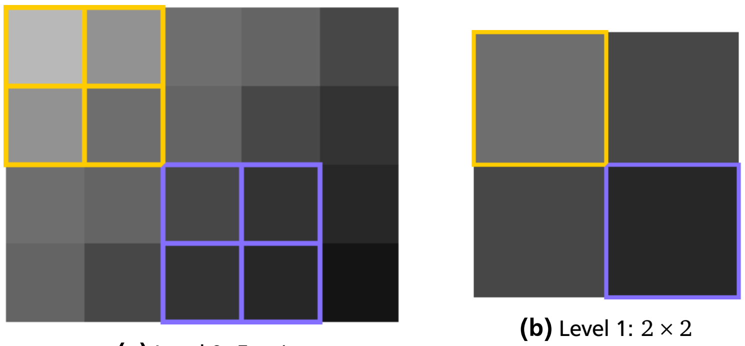

(a) Level 0: 5 × 4 **(b)** Level 1: 2 × 2

Figure 4.2: Hi-Z approximation for NPOT textures with undersized range of minimal Z values. **The failing exacerbates in each iteration.** Colored squares illustrate source and destination for texel fetches.

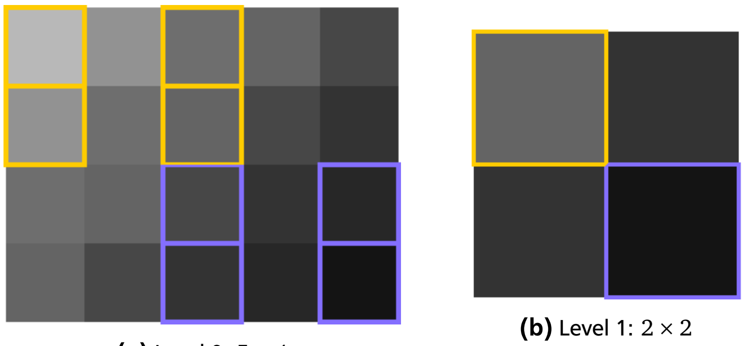

(a) Level 0: 5 × 4 **(b)** Level 1: 2 × 2

Figure 4.3: Enhanced Hi-Z approximation for NPOT textures. **Captures a wider range of minimal Z values.** Colored squares illustrate source and destination for texel fetches.

4.2.5 Pre-Integration Pass

In the pre-integration pass we generate the *visibility* buffer for proper integration in the cone tracing process. This buffer stores the information how much visibility the Hi-Z buffer captures in each *cell*.

Definition 2 (cell) *A cell here denotes the abstraction of a texel at a specific MIP level.*

The visibility is required during cone tracing to determine how much the cone intersects the geometry at each cell. The input for this pass is our Hi-Z buffer and the visibility buffer is generated in a hierarchical fashion, too. At the root level (the first MIP-map) each depth texel has a 100% visibility. As we go up in this hierarchy, the total visibility for the coarse representation of the cell has less or equal visibility to the four finer texels ($Visibility_n \leq Visibility_{n-1}$). Think about the coarse depth cell as a volume containing the finer geometry. Our goal is to calculate how much visibility we have at the coarse level. The cone tracing pass will then sample this visibility at various levels and use it as weight for the incident light color until we accumulate a visibility of 100%. An 8 bit per channel texture is enough for this buffer, which provides 256 steps for visibility.

In this pass we need linear depth values. This is because the visibility factors are computed linearly and the precision must be distributed linearly as well, since we use an 8 bit integral texture format. Although we provide a post-projected depth buffer only, we can transform this to a linear depth value casually with the following function:

$$Linearize(z) = \frac{2 \times P_{near}}{P_{far} + P_{near} - z \times \left(P_{far} - P_{near}\right)} \tag{4.2}$$

P_{near} denotes the near clipping plane and P_{far} denotes the far clipping plane, which is passed to the shader as input parameters. Note that such transformations depend on the used projection matrices. This is important especially when dealing with OpenGL *and* Direct3D. In our case we use *left-handed* projection matrices in the pixel shaders, which is the default for Direct3D. Alternatively, one can multiply the vector $(0, 0, z)^T$ with the inverse projection matrix, which has a larger memory footprint.

The following shader code shows the main function for this pass:

```
// Fetch texels from previous depth map LOD.
ivec2 coordCoarse = ivec2(gl_FragCoord.xy);
ivec2 coordFine   = coordCoarse * ivec2(2);

int mipPrevious = mipLevel - 1;

vec4 fineZ;
fineZ.x = Linearize(texelFetch(depthMap, coordFine              , mipPrevious).r);
fineZ.y = Linearize(texelFetch(depthMap, coordFine + ivec2(1, 0), mipPrevious).r);
fineZ.z = Linearize(texelFetch(depthMap, coordFine + ivec2(0, 1), mipPrevious).r);
fineZ.w = Linearize(texelFetch(depthMap, coordFine + ivec2(1, 1), mipPrevious).r);

// Fetch fine visibility from previous visibility map LOD.
vec4 visibility;
visibility.x = texelFetch(visibilityMap, coordFine              , 0).r;
visibility.y = texelFetch(visibilityMap, coordFine + ivec2(1, 0), 0).r;
visibility.z = texelFetch(visibilityMap, coordFine + ivec2(0, 1), 0).r;
visibility.w = texelFetch(visibilityMap, coordFine + ivec2(1, 1), 0).r;

// Integrate visibility.
float maxZ = max(max(fineZ.x, fineZ.y), max(fineZ.z, fineZ.w));
vec4 integration = visibility * (fineZ / maxZ);

// Compute coarse visibility (with SIMD 'dot' intrinsic).
// This is the output value for the current render target.
float coarseVisibility = dot(vec4(0.25), integration);
```

A further input parameter for this shader is *mipLevel* which specifies the current MIP level we are writing to. Again: at the root level the visibility is 100% so the texel values of the original visibility buffer are all set to 1 (see Figure 4.4).

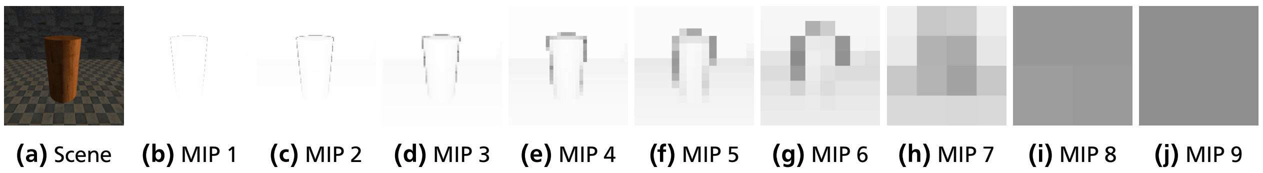


Figure 4.4: Entire MIP chain of the visibility buffer for a 512 × 512 resolution. *MIP 0* is omitted which is completely white.

As the interseted reader may have noticed, we are facing the same problem as in the Hi-Z pass: NPOT resolutions. Fortunately the resulting inaccuracy for the pre-integration pass is not that conspicuous as for the Hi-Z pass. While the accuracy of the Hi-Z buffer is crucial, the visibility buffer is used for a rough approximated integration. Therefore we do not use any offsets for sampling here.

4.2.6 Pre-Convolution Pass

In the pre-convolution pass we merely generate the pre-filtered MIP-maps of the color buffer. This can be done with the default MIP-map generation process by the GPU or with a manual *gaussian blur*. We leave it with the standard GPU based process for performance reasons.

4.2.7 Ray-Tracing Pass

Now we reached the ray tracing pass, which is possibly the most complex part. We will separate this pass in several procedures and analyse the most important of them. However, we will not look at the entire shader since it is too large to explain each line of code.

Recall the fact that we can interpolate the depth values in screen space since we use a post-projected depth buffer (see 4.2.1). That means we can parameterize our ray tracing function:

$$Ray(R,z) := R_{origin} + R_{direction} \times z \tag{4.3}$$

R is then our reflection ray and z is the depth value to intersect the new ray with the specified depth plane. So instead of 'moving' along the ray direction we set its position to a specific depth value in each iteration. But to do this we first need to normalize the ray so that $R_{direction_z} = 1$ and $R_{origin_z} = 0$ holds true:

$$\begin{aligned} R_{direction} &:= \frac{V_{SS}}{V_{SS_z}} \\ R_{origin} &:= P_{SS} + R_{direction} \times -P_{SS_z} \end{aligned} \tag{4.4}$$

The former equation normalizes the *ray direction* where V_{SS} is the direction of the original reflection ray and the latter equation locates the *ray origin* to the *near clipping plane* where P_{SS} is defined as $P_{SS} := \left(TexCoord_{xy} \quad depth\right)$; both in screen space. With this parameterization we can easily interpolate the position along the ray between the *near*- and *far clipping planes*.

Now it should be clear why HZCT is not able to ray trace towards the camera. If V_{SS_z} is nearly zero the computation for $R_{direction}$ gets very unprecise due to *IEEE-754 floating point arithmetic* [IEEE, 1985]. In addition the ray setup is intended to point from the *near*- to the *far clipping plane* and not vice versa. So that means the algorithm can not find all ray intersections which could actually be reconstructed from screen space. Fortunately the most reflections we are interested in can be captured with HZCT (see Figure 4.5). Moreover we can augment this technique with a *linear ray march*, which we have already seen.

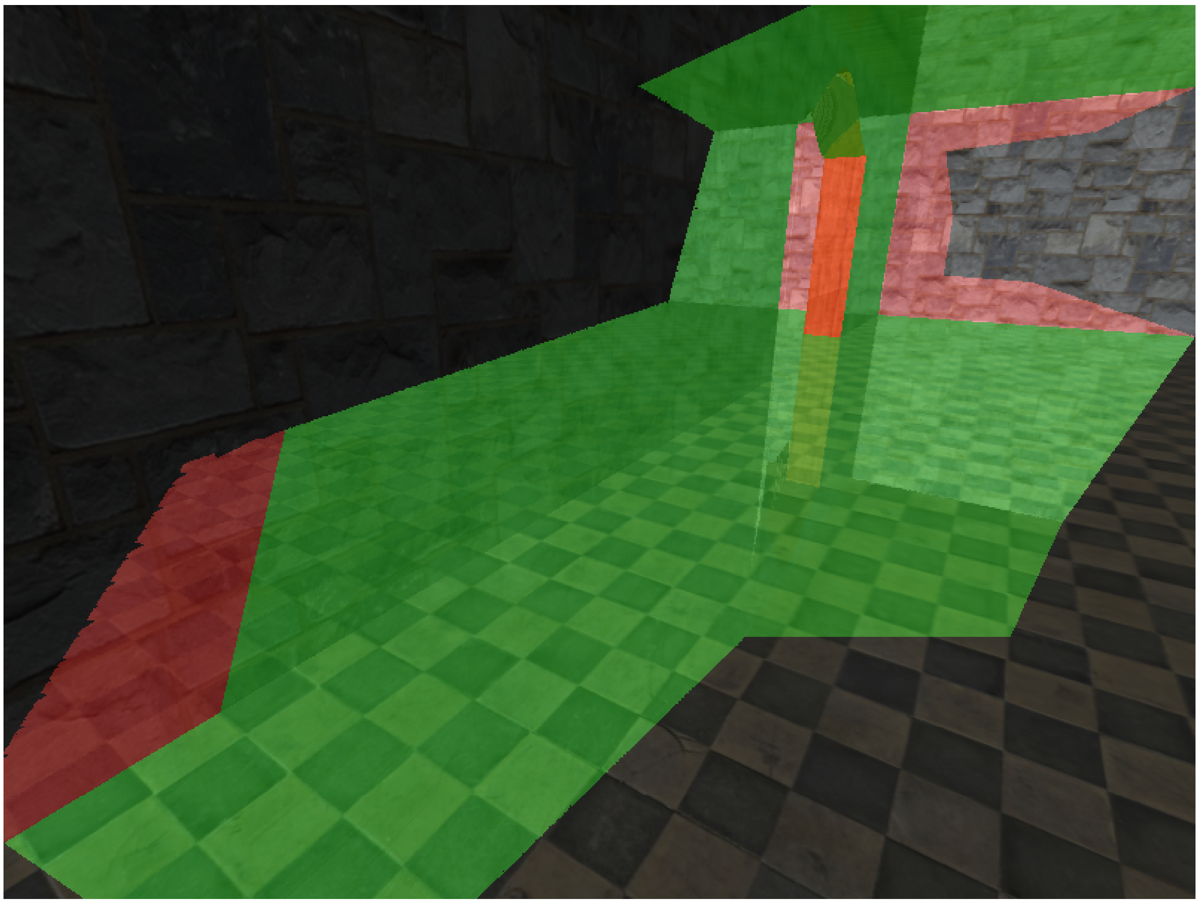


Figure 4.5: Illustrates the distribution of HZCT for rays pointing away from the camera (in green) and *linear ray marching* for rays pointing towards the camera (in red).

Once we have a normalized screen space reflection vector, we can run the Hi-Z ray tracing procedure (see Algorithm 4 for the core idea). The variable *level* stores the current MIP level from which we sample the depth values of the Hi-Z buffer. Whenever the ray intersects the boundary depth plane we go up a level and when the ray intersects the minimum depth plane we go down a level. As soon as *level* falls below zero we found the exact ray intersection since there is nothing more to refine. See Figure 4.6 for illustration of the ray iterations.

The most complex task in this pass is to determine if the ray crossed a cell boundary. The pseudocode only shows a rough guideline, so to better understand how this determination can be performed on the GPU, take a look at the following code sample:

Algorithm 4 **Ray Tracing Algorithm** for HZCT.

```
1: procedure HiZRayTrace($R$)
2:     $level \leftarrow 0$
3:     while $level \geq 0$ do
4:         $minimumPlane \leftarrow$ GetCellMinimumDepthPlane(...)
5:         $boundaryPlane \leftarrow$ GetCellBoundaryDepthPlane(...)
6:         $closestPlane \leftarrow \min\{minimumPlane, boundaryPlane\}$
7:         $ray \leftarrow$ Ray($R, closestPlane$)                    ▷ Intersect the closest plane
8:         if IntersectedMinimumDepthPlane(...) then
9:             $level \leftarrow level - 1$                              ▷ Go down a level
10:        if IntersectedBoundaryDepthPlane(...) then
11:            $level \leftarrow level + 1$                              ▷ Go up a level
12:    return $ray$                                                  ▷ Return intersection ray position
```

```glsl
// Parameterized ray-tracing procedure. 'z' must be in the interval [0, 1].
vec3 Ray(vec3 rayOrigin, vec3 rayDirection, float z) {
    return rayOrigin + rayDirection * z;
}

// Returns the number of cells for the specified texture size and MIP level.
vec2 GetCellCount(vec2 size, float level) {
    return floor(size / (level > 0.0 ? exp2(level) : 1.0));
}

// Converts the ray position to a cell index.
vec2 GetCell(vec2 ray, vec2 cellCount) {
    return floor(ray * cellCount);
}

// Check if cell index A is equal to cell index B.
bool CrossedCellBoundary(vec2 cellIndexA, vec2 cellIndexB) {
    return
        ( cellIndexA.x != cellIndexB.x ) ||
        ( cellIndexA.y != cellIndexB.y );
}

// This function computes the new ray position where
// the ray intersects with the specified cell's boundary.
vec3 IntersectCellBoundary(
    vec3 rayOrigin, vec3 rayDirection,
    vec2 cellIndex, vec2 cellCount,
    vec2 crossStep, vec2 crossOffset)
{
    // Determine in which adjacent cell the ray resides.
    // The cell index always refers to the left-top corner
    // of a cell. So add "crossStep" whose components are
    // either 1.0 (for positive direction) or 0.0 (for
    // negative direction), and then add "crossOffset" so
    // that we are in the center of that cell.
    vec2 cell = cellIndex + crossStep;
    cell /= cellCount;
    cell += crossOffset;

    // Now compute interpolation factor "t" with
    // this cell position and the ray origin.
    vec2 delta = cell - rayOrigin.xy;
    delta /= rayDirection.xy;

    float t = min(delta.x, delta.y);

    // Return final ray position
    return Ray(rayOrigin, rayDirection, t);
}
```

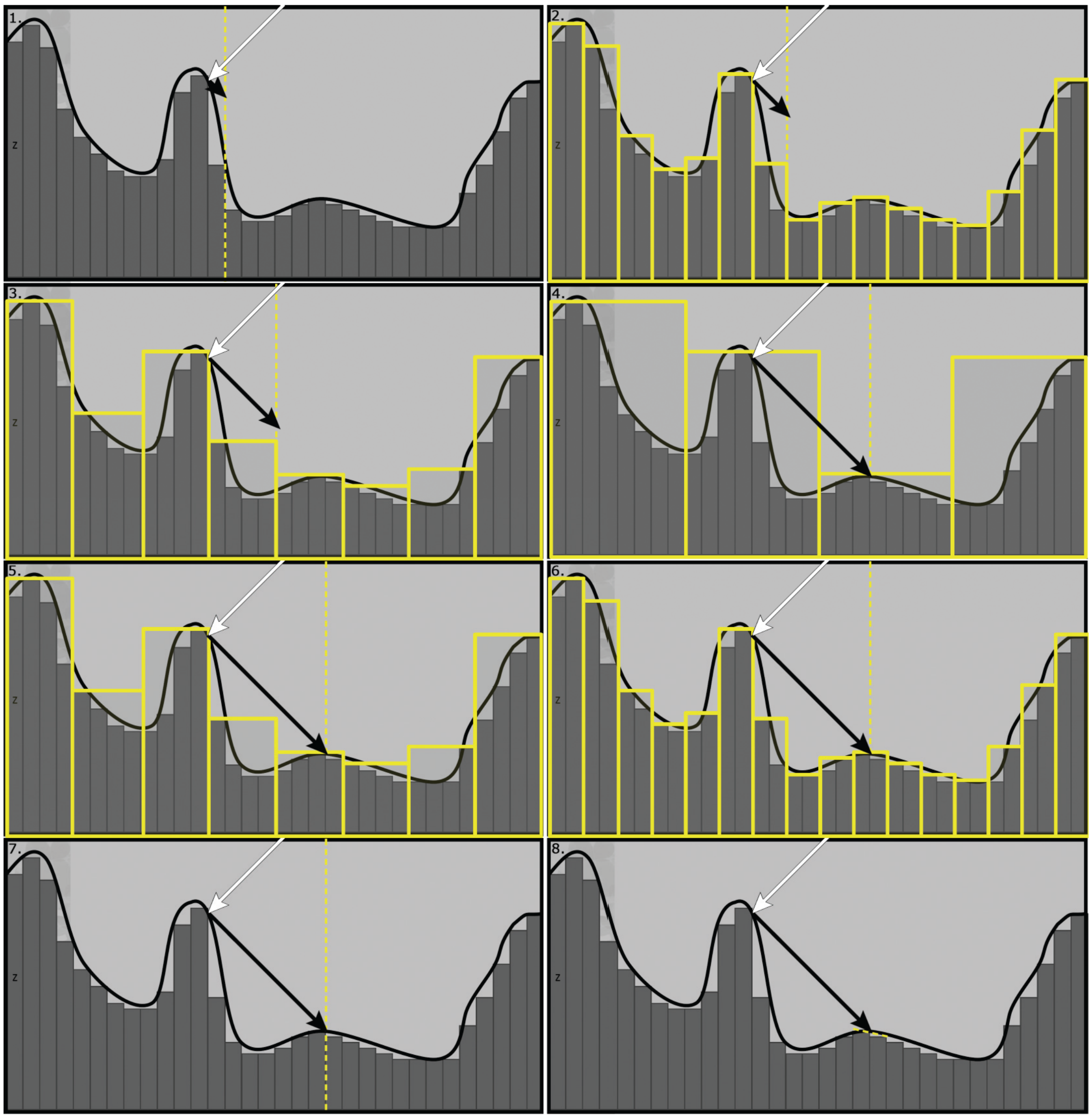


Figure 4.6: Illustration of the HiZRayTrace procedure. *level* is going up and down in the hierarchy to take longer jumps in each iteration. *Gray* bars represent the Hi-Z buffer; *Yellow* bars represent the *cell boundaries*; *White* arrow represents the view ray; *Black* arrow represents the parameterized reflection ray.
Image courtesy of *GPU Pro 5*.

Now to determine if the ray indeed crossed a cell boundary, we compute a new temporary ray position with the IntersectCellBoundary procedure and compare the cell indices with CrossedCellBoundary from the above code sample. It is additionally important to sample the depth buffer with a *nearest* sampling filter. Otherwise the depth values for our cells will be interpolated, which causes erroneous results.

That is the main part for Hi-Z ray tracing, which determines our ray intersections.

4.2.8 Cone-Tracing Pass

In the cone-tracing pass we accumulate all incident light from the color buffer by interpolating between the ray origin and intersection point we got from the ray-tracing pass. To do this we make use of all our hierarchical buffers we have generated so far.

As mentioned earlier to trace a cone in screen space we abstract it by an isosceles triangle (see Figure 4.7). This is again a rough approximation since we don't integrate the entire cone. Nevertheless, we need an angle for our cone. The input parameter we have for a cone angle θ, is the surface roughness ϱ and since the calculation of the reflection ray is

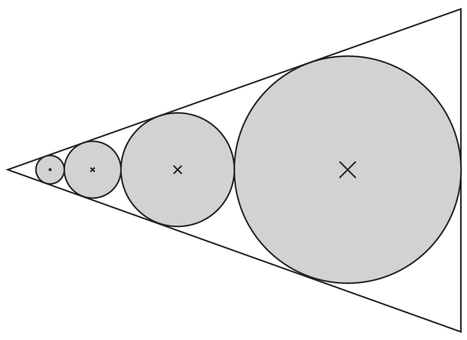


Figure 4.7: Cone abstraction as isosceles triangle. The texture lookups during cone tracing are represented as circles inside the triangle and their radii specify the MIP level. Image courtesy of *GPU Pro 5*.

merely based on the *Phong reflection model*, we can simply convert the roughness to a specular power factor, to make use of the following equations for importance sampling:

$$\begin{aligned}\alpha &= \alpha_{min} + (\alpha_{max} - \alpha_{min}) \times \varrho \\ \theta &= \cos\left(\xi^{\frac{1}{\alpha+1}}\right) \\ \xi &\in [0,1]\end{aligned} \tag{4.5}$$

α denotes the *specular power factor* and ξ denotes a value in the range $[0,1]$. With importance sampling ξ is choosen randomly in the range $[0,1]$ to generate random ray directions within the specular lobe in spherical coordinates [Lawrence, 2002]. But in our case we need only a single sample to cover the specular lobe for our cone. Hence we need a well-adjusted value and 0.244 seems to be a good choice in practice. The interested reader is referred to the book *GPU Pro 5* [Uludag, 2014] for more information about the above formula. For a perfect specular reflection the specular power in the Phong model would need to be infinite, so we can interpolate θ to zero if α is greater than 1000 for instance.

With some trigonometry the radii of the circles can be computed easily. To calculate the in-radius (radius of a circle inside the triangle touching all three sides) of an isosceles triangle, this equation can be used:

$$\begin{aligned}a &= 2h \times \tan(\theta) \\ r &= \frac{a\left(\sqrt{a^2+4h^2}-a\right)}{4h}\end{aligned} \tag{4.6}$$

a denotes the base of the isosceles triangle and is computed with the tangents of a right triangle (the half of the isosceles triangle), r denotes the in-radius of the largest circle which fits into the triangle, and h denotes the height of the triangle (the length of our reflection ray). With the radius of the first circle for our texture sample we can move forwards by simply subtracting twice the radius from the length of our reflection ray, which will procude a new and smaller isosceles triangle. We do this repeatedly until we are finished.

Now that we know how to calculate the radii and circle positions we start integrating the light by sampling the color buffer at the circle centers, where the radii specify at which hierarchy level to read. We use the hardware accelerated *trilinear* sampling filter to interpolate the colors also for the continuous MIP levels. Now these color samples must be weighted depending on how much the cone intersects the geometry. For this we use the visibility- and Hi-Z buffers. If we accumulated a visibility of 100% we can stop the cone tracing process prematurely. This means the larger the cone angle the faster the cone tracing, because we can take less texture samples and from higher MIP levels.

To summarize this section take a look at the following pseudocode (see Algorithm 5). In practice, a good number of maximal iterations in the HiZConeTrace procedure is 7 for a balanced output in quality and performance, i.e. a *for*-loop is used instead of a *while*-loop and the early exit is implemented in the loop body with a *break* instruction. The roughness ϱ, to calculate the angle θ, can be determined by sampling the screen space at the pixel's location from an additional texture such as a *glossy*-, *specular*-, or *roughness* map. Otherwise a fixed value from a shader input parameter can be used.

4.2.9 Fading on Screen Border

We have finally reconstructed reflections from screen space. But we know that the screen space does not cover everything which is actually reflectable. So we have to consider what happens when our ray tracing pass fails, because the ray

Algorithm 5 **Cone Tracing Algorithm** for HZCT.

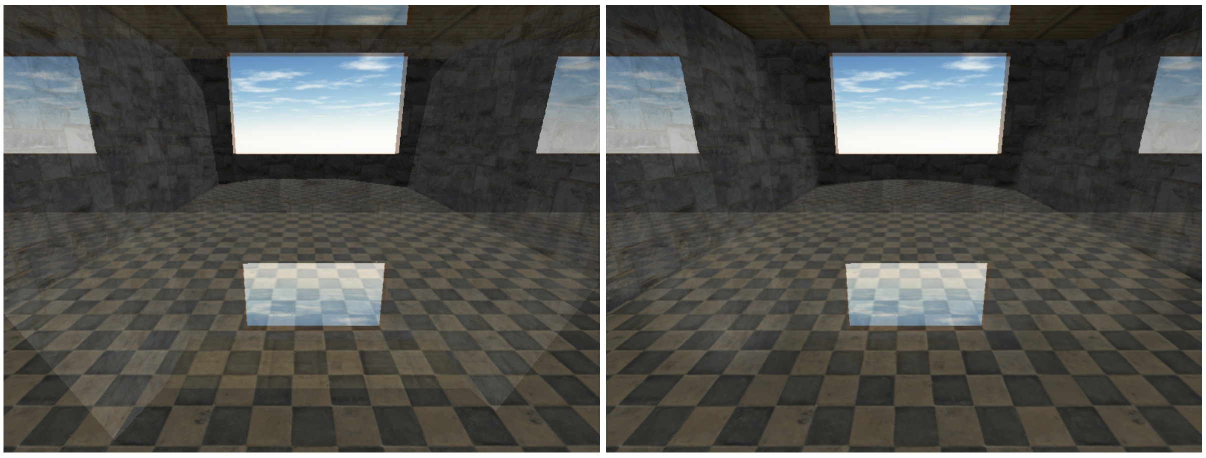
```
 1: procedure HiZConeTrace($R, \theta$)
 2:     $h \leftarrow \|R_{direction}\|$                          ▷ Initialize $h$ to length of reflection ray
 3:     $\delta \leftarrow$ Normalize($R_{direction}$)              ▷ Get normalized ray direction $\delta$
 4:     $C \leftarrow (0,0,0,0)$                 ▷ Initialize reflection color and use alpha channel for visibility
 5:     while $C_{alpha} < 1$ do                      ▷ Trace cone until the visibility reached 100%
 6:         $a \leftarrow$ IsoscelesTriangleBase($h, \theta$)                 ▷ Calculate triangle base $a$
 7:         $r \leftarrow$ IsoscelesTriangleInRadius($h, a$)                  ▷ Calculate in-radius $r$
 8:         $P_{sample} \leftarrow R_{origin} + \delta \times (h - r)$             ▷ Calculate sample position
 9:         $l_{MIP} \leftarrow \log_2\left(r \times \max\{Resolution_x, Resolution_y\}\right)$   ▷ Convert in-radius into screen-space MIP level
10:         $C \leftarrow C +$ ConeSampleWeightedColor($P_{sample}, l_{MIP}$)   ▷ Accumulate color with hierarchical buffers
11:         $h \leftarrow h - r \times 2$                          ▷ Reduce height to move forwards
12:     return $C/C_{alpha}$                              ▷ Return normalized reflection color
```

intersection lies outside the screen. We would have a reflection which is cropped at its edges. Such edges will stand out visibly since our eyes are trained to detect borders, ultimately destroying the illusion of local reflections.

A good and easy method to hide away this flaw is to fade-out the reflection at the screen border. A simple fading function can be implemented like this:

$$\begin{aligned} I_{end} &= 1 \\ I_{start} &\in (0, I_{end}] \\ D_{boundary} &= \left\| R_{intersection_{xy}} - (0.5, 0.5)^T \right\| \times 2 \\ f_{border} &= 1 - \max\left(0, \min\left(\frac{D_{boundary} - I_{start}}{I_{end} - I_{start}}, 1\right)\right) \end{aligned} \tag{4.7}$$

I_{start} specifies the interpolation start factor, which is set to 0.8 in the demo and $R_{intersection}$ specifies the ray intersection in screen space. The nested min and max functions are used to clamp the fading to the range $[0, 1]$. The final reflection color is then multiplied by f_{border}. A comparison between the fading being on and off is shown in Figure 4.8.



(a) Fading Off **(b)** Fading On

Figure 4.8: Sharp edges are clearly visible on the floor in Figure 4.8a. Reflections are smoothly faded out at the floor in Figure 4.8b.

4.2.10 Fallback: Linear Ray March

As we have seen, HZCT is unable to cope with rays which are directed back into the direction of the camera. In this section, we enhance HZCT with a fallback method to solve this case.

We can simply adjust our final ray tracing procedure with a conditional branch at its top (see Algorithm 6). A possible implementation of LinearRayMarch has already been shown in Section 4.2.2. If $R_{direction_z}$ is less than or equal to the view direction ϵ, we try to find an intersection with linear ray marching. The original method in *GPU Pro 5* merely appends

Algorithm 6 Fallback to linear ray march for HZCT.

```
1: procedure RAYTRACE($R$)
2:     if $R_{direction_z} \leq \epsilon$ then        ▷ Check if the ray points towards the camera
3:         LINEARRAYMARCH($R$)                         ▷ Use fallback method
4:     else
5:         HIZRAYTRACE($R$)                            ▷ Use primary ray tracing method
```

a further fading mechanism to avoid rays pointing towards the camera. With our fallback we are able to extract more information from the screen space (a visualization is shown in Figure 4.5).

However, this fallback may cause a noteworthy disadvantage: *dynamic branching*. Modern GPUs are highly parallelized processors with tight Single Instruction Multiple Data (SIMD) design, which makes it difficult to utilize it efficiently with incoherent instruction executions, i.e. when contiguous shader units execute different program paths. There are various publications about improving the utilization of SIMD CPUs and GPUs, particularly for ray tracing (*Faster incoherent rays: Multi-BVH ray stream tracing* [Tsakok, 2009], *Improving SIMD efficiency for parallel Monte Carlo light transport on the GPU* [van Antwerpen, 2011]). Nevertheless, our fallback here may slow down the ray tracing pass when contiguous pixels deviate in their execution paths. This is due to the very different ways the procedures LINEARRAYMARCH and HIZRAYTRACE work.

4.2.11 Additional Extensions

Now we actually completed the implementation of SSCT but there are several prospects for additional extensions. One of them is to add a further fallback such as an *environment map* for the sky for instance (here, the cube map is also called a *skybox*). In this case the sampling and cone tracing works completely different. A good solution for cube maps would be to pre-filter it with a blur pass. In the final reflection shader we can then fade-in the skybox reflections when the local reflections are faded-out.

Another extension would be to compute several bounces to simulate mirrors, which are facing each other. It could also increase the plausibility of indirect lighting in general.

Yet another extension would be to make use of *temporal filtering* by taking the previously rendered frame into account. This may improve performance and image stability. If the result of a ray traced color buffer is available to the next frame, the neighboring pixel information can be fetched to reduce artifacts.

4.3 Competitor: SSLR

Before we compare SSCT with SSLR methods we first take a closer look at glossy reflections in SSLR, such as *Killzone Shadow Fall* [Valient, 2014]. For the further proceeding we will speak only of *Killzone* if we mean *Killzone Shadow Fall*. Although SSLR only uses ray tracing there are several methods to implement glossiness. The naïve approach is to cast multiple rays and take the mean value of the color samples. A smarter approach is to split the algorithm into multiple passes again and blur the ray trace color buffer. This is how it is done in *Killzone* (see Section 3.1.5). A mask buffer is used to enhance the blurring of the ray trace color buffer. We can divide this algorithm into the following three steps:

1. **Ray-Tracing Pass** generates the ray trace color buffer (see Figure 3.5) and mask buffer.

2. **Blur Pass** generates the MIP levels for both the color- and mask buffer with a gaussian blur.

3. **Reflection Pass** draws final reflections on the screen.

4.3.1 Uniform Ray Marching

The linear ray march (or rather the ray-tracing pass) in *Killzone* is exceptionally simple. Only a constant step size is used and even a binary search for refinement is omitted. The uniform step size is computed once for every pixel, depending on the surface roughness. The higher the roughness the larger the step size. Thereby a higher roughness results in faster detection of ray intersections, but at the same time the image quality does not suffer, due to stronger blurring. This therefore means that the roughness also determines the MIP level of the blurred ray tracing color buffer — similar to the MIP selection in the SSCT cone tracing pass.

This pass will now generate the *ray trace color buffer*. This means that we do not store the information about the ray intersection. Solely the data from the original color buffer is written to the shader output. Otherwise the blur pass would be unnecessary. The second shader output is the mask buffer, to which we only write *one* if we found a ray intersection and *zero* otherwise. This will produce a monochrome image, which is then transformed into a grayscale image with the blur pass.

4.3.2 Gaussian Blur

Soft, blurry images are commonly generated with a *gaussian blur*, which we have already mentioned. In SSCT a blur pass to the color buffer is optional because we can also use the default generated MIP maps. But to achieve glossiness in SSLR the blurred color buffer is crucial.

A gaussian blur can be implemented as an entirely separated and general purpose post-processor. Though a general purpose blur pass may be more inefficient instead of a fixed count of 'taps'. In *Killzone* a "7 tap horizontal/ vertical separable blur" is used [Valient, 2014]. This means 7 texture samples are fechted for both horizontal and vertical blur passes. A gaussian blur itself is divided into two render passes: one to blur the image horizontal and then vertical (or vise versa). This reduces the number of texture samples drastically. The weights for the texture samples are calculated with the *gaussian normal distribution function*, or rather with its one-dimensional Probability Density Function (PDF):

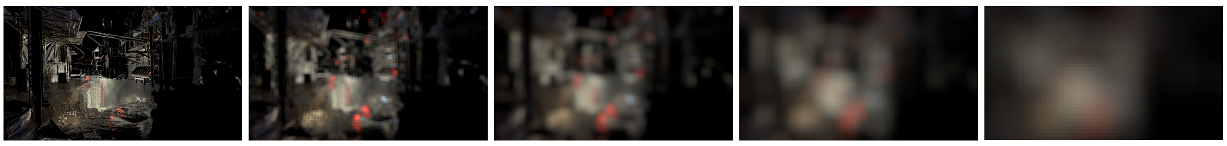
$$\begin{aligned} f_{\sigma,\mu}(x) &= \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \\ \mu &\in \mathbb{R} \\ \sigma &\in \mathbb{R}^+ \end{aligned} \tag{4.8}$$

This function describes the normal distribution. We can use this for our sample weights to get a smooth blurry image. For a gaussian blur, μ (the mean value) is always zero and σ (the variance) is commonly set to one. As already stated we use a fixed count of samples, thus we can pre-compute the sample weights and hard-code it into the blur shader. This may look like the following example of a *horizontal* blur shader:

```
1  vec4 color = vec4(0.0);
2  color += texture(tex, texCoord + vec2(invTexWidth *  3.0, 0)) * 0.002;
3  color += texture(tex, texCoord + vec2(invTexWidth *  2.0, 0)) * 0.028;
4  color += texture(tex, texCoord + vec2(invTexWidth *  1.0, 0)) * 0.233;
5  color += texture(tex, texCoord                             ) * 0.474;
6  color += texture(tex, texCoord + vec2(invTexWidth * -1.0, 0)) * 0.233;
7  color += texture(tex, texCoord + vec2(invTexWidth * -2.0, 0)) * 0.028;
8  color += texture(tex, texCoord + vec2(invTexWidth * -3.0, 0)) * 0.002;
```

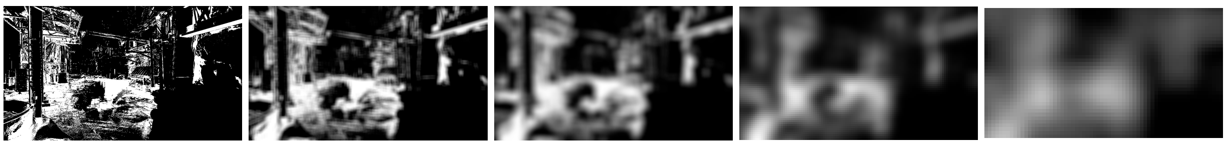
color denotes the final image color, *tex* denotes the texture object, *texCoord* denotes the current pixel coordinate (in texture space), and *invTexWidth* denotes the inverse texture width. The sum of all weights is exactly 1. This ensures that the blurred image is not brighter or darker, just blurred. When the weights are calculated dynamically and with varying counts, the weights must be normalized by dividing them by their sum. This is because we compute a *discrete* number of weights with a *continuous* density function.

The above code sample shows a simple horizontal gaussian blur pass. In *Killzone* the previously generated mask buffer is used additionally to weight the texture samples. If a 'miss' is read from the mask buffer (i.e. the value zero), then always the central pixel is used (in this case line number 5). This enhances the image quality of the ray trace color buffer MIP chain. Such an example MIP chain is shown in Figure 4.9 and Figure 4.10.



(a) MIP 0 **(b)** MIP 1 **(c)** MIP 2 **(d)** MIP 3 **(e)** MIP 4

Figure 4.9: First five MIP levels of the ray trace color buffer.



(a) MIP 0 **(b)** MIP 1 **(c)** MIP 2 **(d)** MIP 3 **(e)** MIP 4

Figure 4.10: First five MIP levels of the mask buffer.

4.3.3 Reflection in SSLR

The final reflection in SSLR is done in the third and last stage. Here we actually only draw the main- and the ray trace color buffer, but from different MIP levels. This is an essential difference to SSCT. Instead of interpolating from the ray origin to the intersection point and integrating all incident radiance, we simply draw the final reflection by choosing the respective level of glossiness in the MIP chain. The spreading of the reflection rays depends on the surface roughness *and* their lengths. In other words, the longer the cone the larger its radius. The length of the ray traversal in the first stage could be written to the alpha channel of the ray trace color buffer. Since an *8-bit unsigned integral RGBA* format is typical for such a texture, we should normalize this length to the interval $[0,1]$ and note that the precision is limited to $2^8 = 256$ values, which is sufficient. An example implementation is shown in the following code sample:

```glsl
// Determine MIP lod for reflection color
float spreading = textureLod(rayTraceColorBuffer, texCoord, 0.0).a;
float mipLod = mipCount * roughness * spreading;

// Sample final scene- and ray-trace colors
vec4 color = texture(colorBuffer, texCoord);
vec3 rayTraceColor = textureLod(rayTraceColorBuffer, texCoord, mipLod).rgb;

// Add reflection color times 'reflectivity' factor
color.rgb += rayTraceColor * reflectivity;
```

color denotes the final pixel color; *mipLod* is calculated by the number of MIP levels denoted by *mipCount*, the surface roughness, and the spreading, stored in the alpha channel of the ray trace color buffer.

4.4 Optimizations

There are many ways for optimizations in post-processing effects. Particular for ray tracing effects with multiple stages. As already mentioned, a frequently used optimization is to render the respective effects only at half resolution. However, this is usually only justifiable when blur passes are involved, which hide the lower texture quality.

Another issue is memory efficiency. Incoherent memory access stalls the graphics pipeline, due to wasted *memory bursts* (larger blocks of data for better cache utilization). The game *Thief 4* approximates the normal vectors with $(0,0,1)^T$ for better memory coalescing [Sikachev and Longchamps, 2014]. The *bump mapping* (effect to enhance shading appearance on textures) is implemented supplementary as a post-process, i.e. the normal deviation is applied after the ray tracing.

Furthermore gradient-based texture operations should be avoided within dynamic branching, or they should be at least moved out of flow control to prevent divergence. They may force the pipeline to load texture data for program paths where they are not needed, due to the massive parallelism on GPUs. In practice, this means that the intrinsics `textureLod`/ `textureGrad` should be prefered over `texture` (in GLSL) and `SampleLevel`/ `SampleGrad` should be prefered over `Sample` (in DirectX High Level Shading Language (HLSL)) respectively.

5 Results and Discussion

It is time to compare the quality and performance of our method to related work. The example images compare our method SSCT with the original method HZCT, then with the related method SSLR (implemented as explained in section 4.3), and finally with the ground truth, rendered with the Mitsuba Renderer [Jakob, 2010]. All rendering times are determined with a hardware timer query, which is very percise and they only reflect the rendering duration of the effect, excluding the scene rendering. The following system setup was used to render the results:

Attribute	Setup
Operating System	Microsoft Windows 7 Professional
CPU	Intel® Core™ i7-3770K @ 3531 MHz
GPU	NVIDIA GeForce GTX 670 (PCIe/SSE2)
RAM	16 GB DDR3
VRAM	2048 MB 256-bit GDDR5
Renderer	OpenGL 4.5.0 Core Profile

First, we will compare SSCT with HZCT. In that context, the first image comparison (see Figure 5.1) shows a wooden cylinder on the floor which reflects the surrounding tiles.

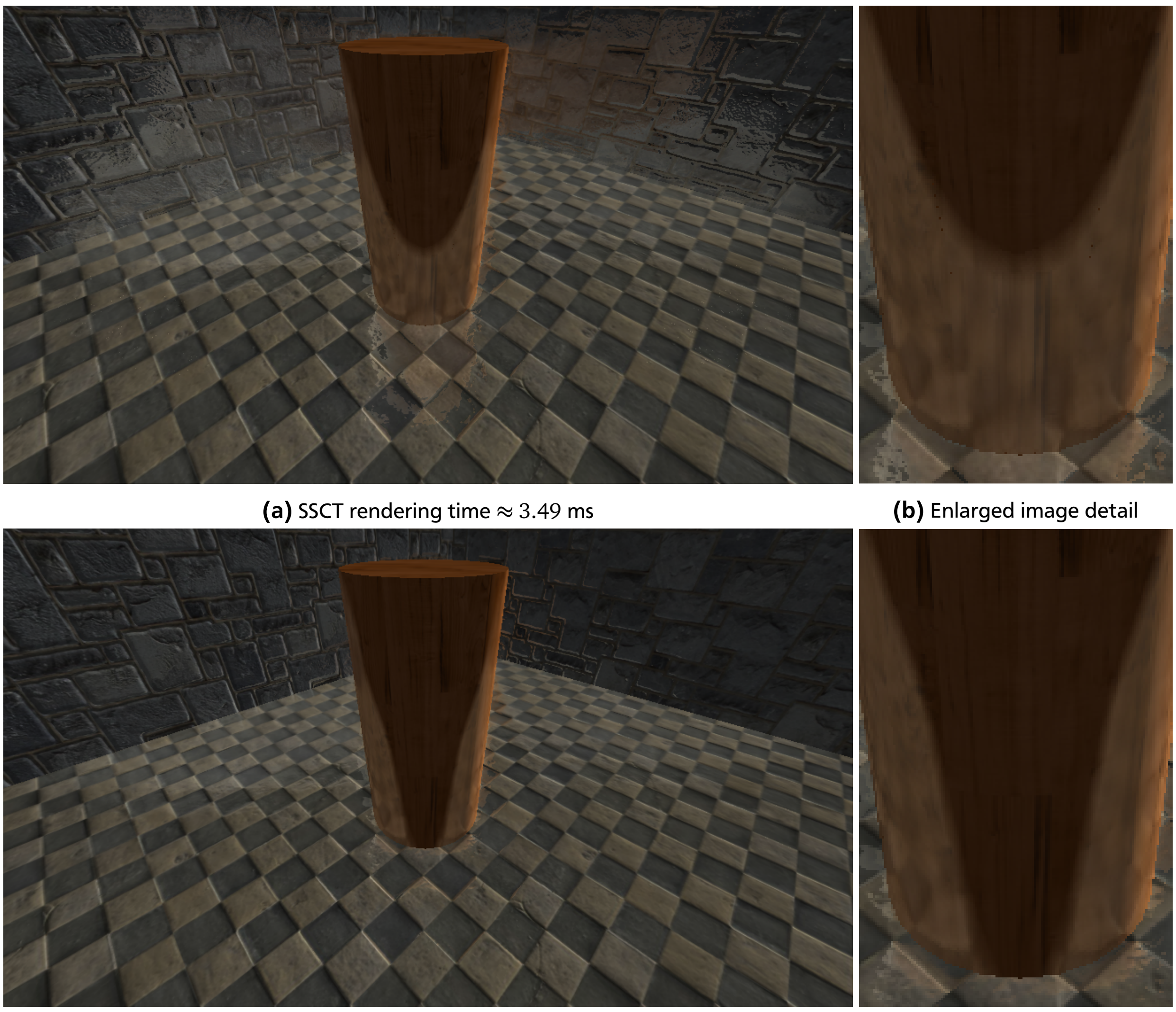
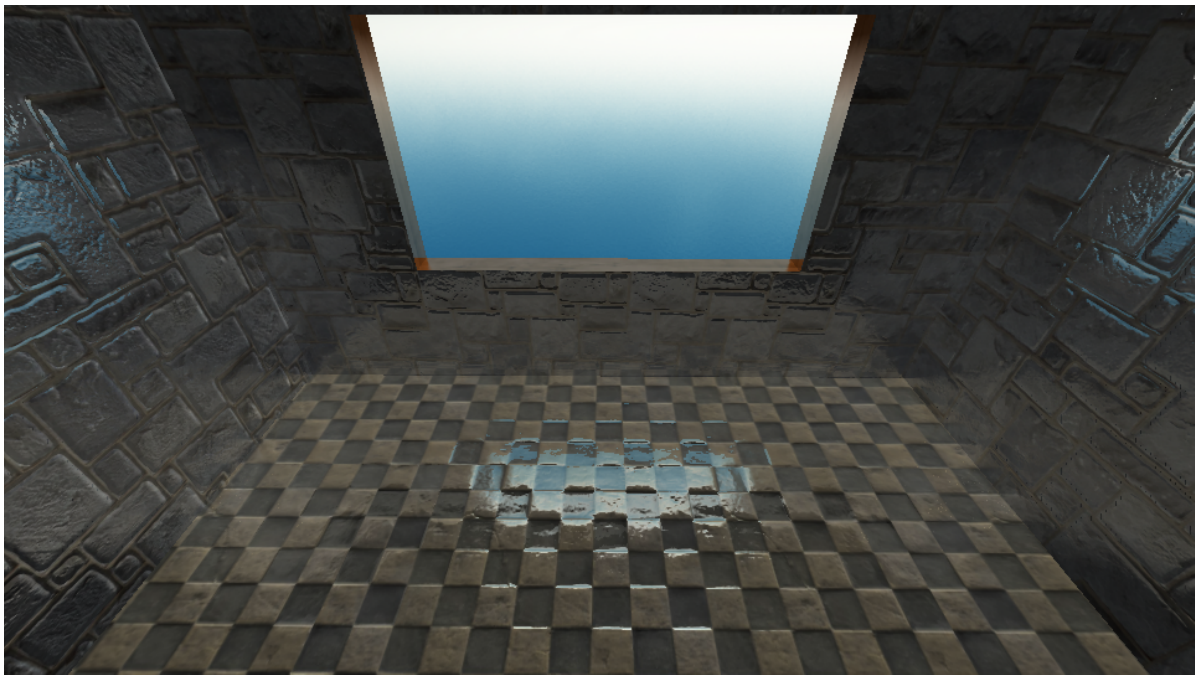
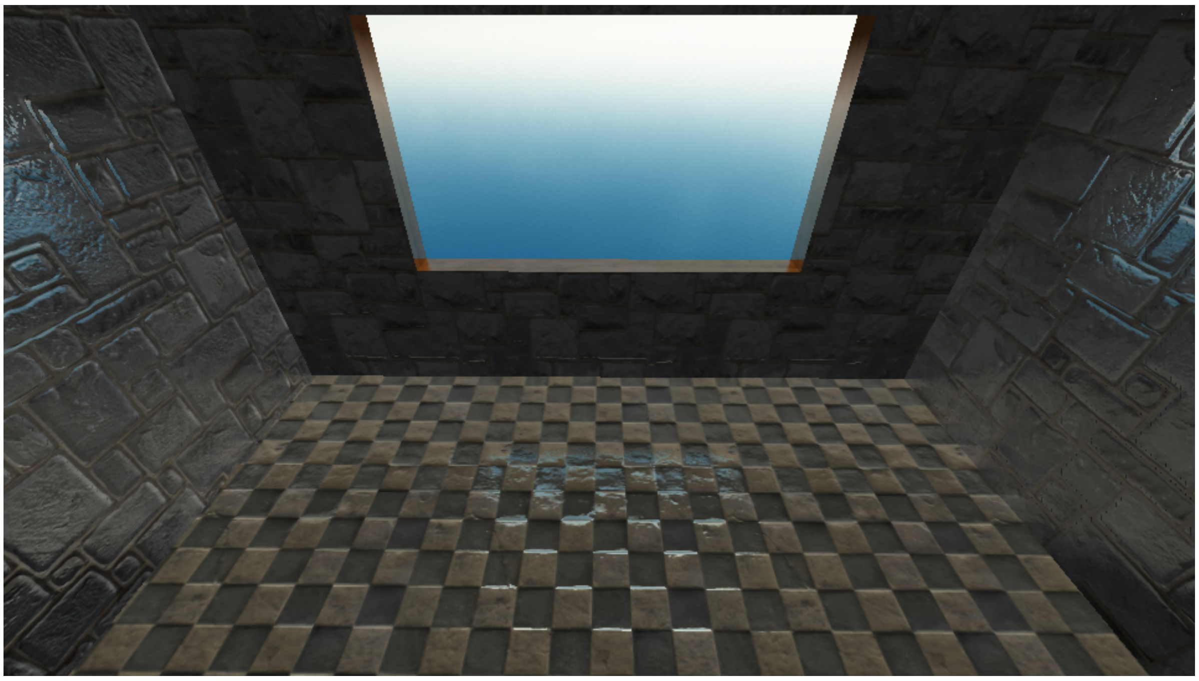

(a) SSCT rendering time ≈ 3.49 ms
(b) Enlarged image detail
(c) HZCT rendering time ≈ 1.04 ms
(d) Enlarged image detail

Figure 5.1: Looking down to the floor at a cylinder, which reflects the tiles (960 × 540 resolution).

In the next image comparison (see Figure 5.2) the outside of a window is reflected on the floor. Here, HZCT only captures a small amount of that window, but SSCT extends the range of the reflective area. Moreover HZCT ignores the relfections on the front wall completely, whereas SSCT additionally reflects the floor underneath the window.



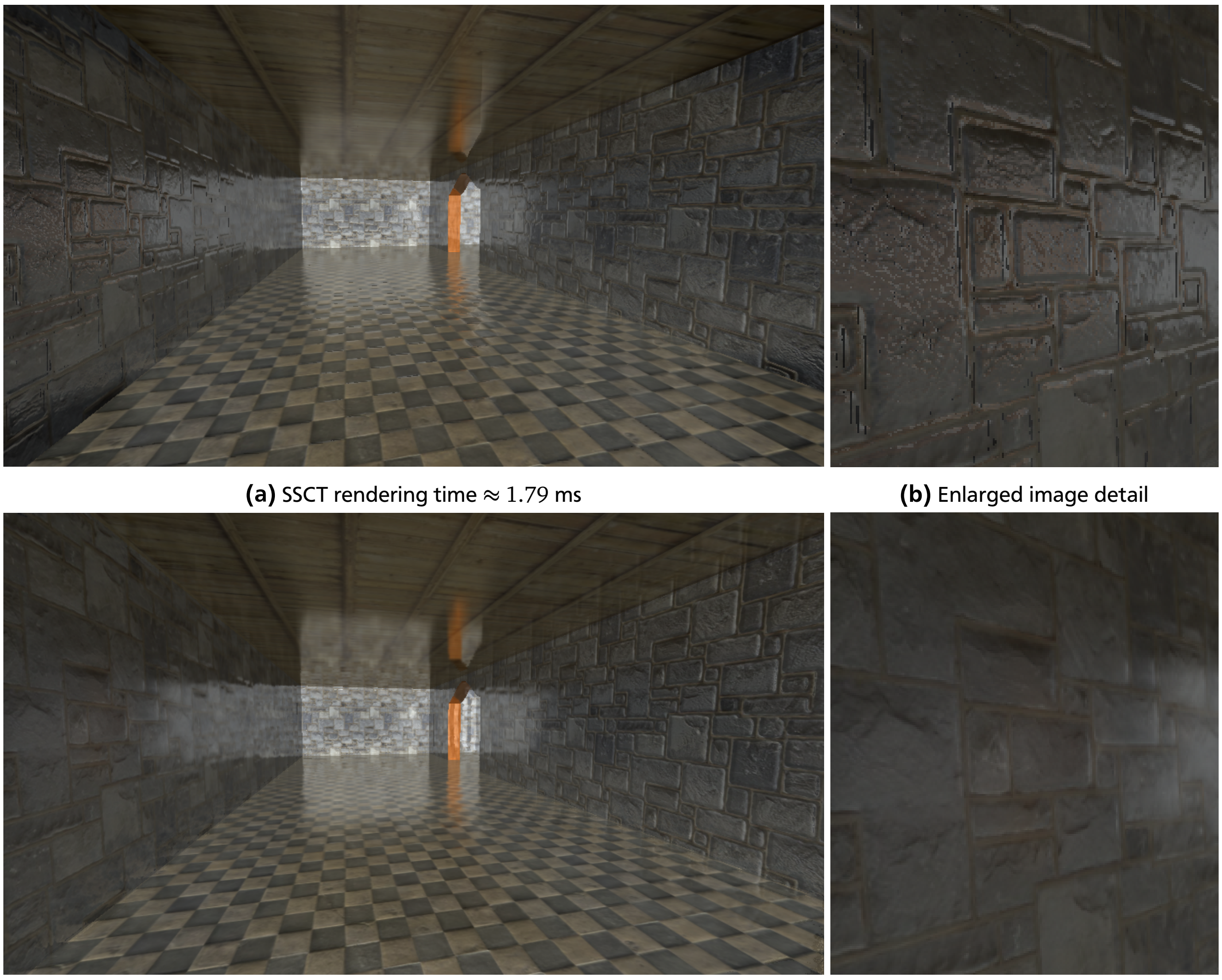
(a) SSCT rendering time ≈ 2.25 ms



(b) HZCT rendering time ≈ 1.12 ms

Figure 5.2: Looking down to the floor, which reflects the outside of a window (960 × 540 resolution).

In the following, we will compare SSCT with SSLR. In that context, the first image comparison (see Figure 5.3) shows a corridor with glossy reflections on the tiled floor, stone walls and wooden ceiling. Only the floor and walls have a normal map. The details are clearer in SSCT as shown in the image enlargement, where a section of the left wall is visible.



(a) SSCT rendering time ≈ 1.79 ms

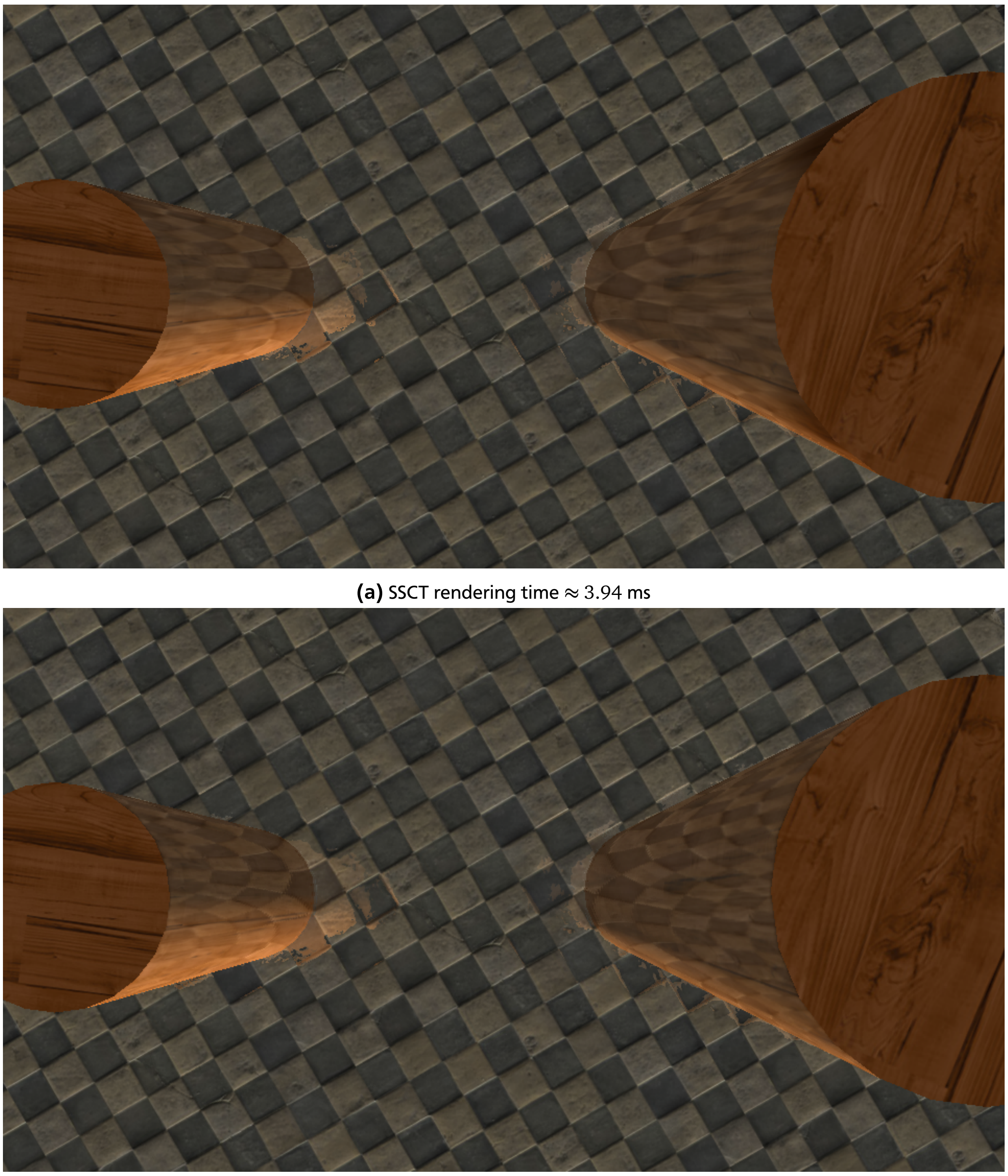
(b) Enlarged image detail

(c) SSLR rendering time ≈ 1.63 ms

(d) Enlarged image detail

Figure 5.3: Looking along a corridor with reflections on the floor, walls, and ceiling (960 × 540 resolution).

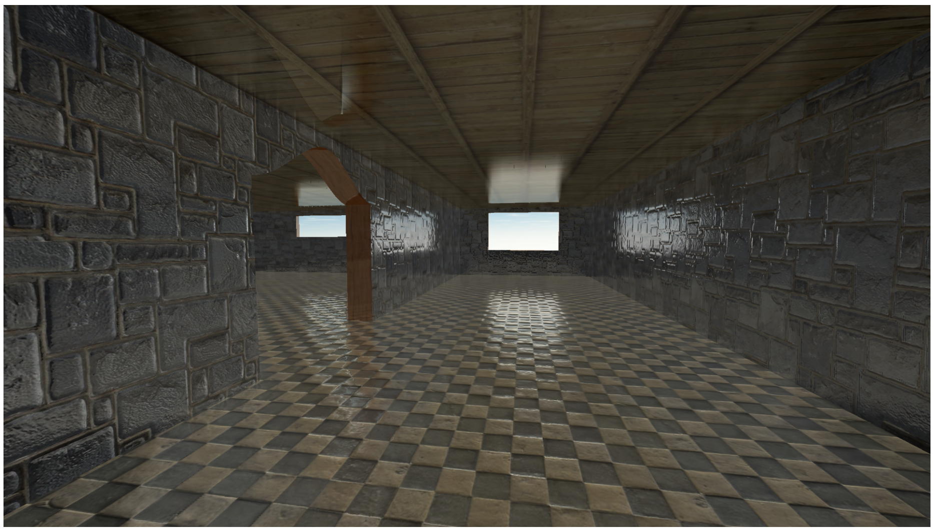
The next image comparison (see Figure 5.4) shows the same scene at a different location. There are two wooden cylinders on the floor which reflect the surrounding tiles. The rough surface of the wood has a more glossy appearance with SSCT, but this may also depend on some parameter fine-tuning for both effects. This is due to the differnet transfers of the BRDF material parameters in each effect. The performance loss of SSCT in this scenario originates from the many rays on the floor which point towards the camera. As we know we can not make use of Hi-Z ray traversal for these rays. So we have lots of rays with low-performance program paths and only a few of them produce a final reflection.

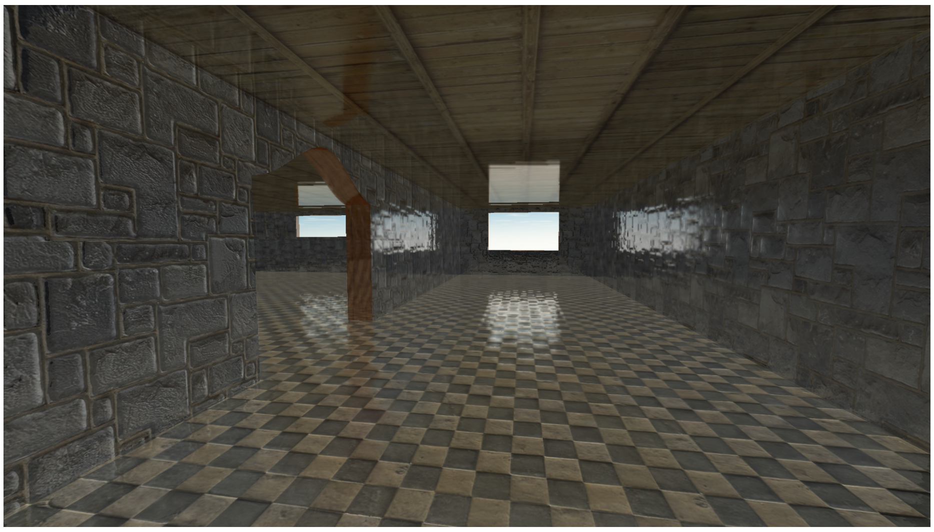

(a) SSCT rendering time ≈ 3.94 ms

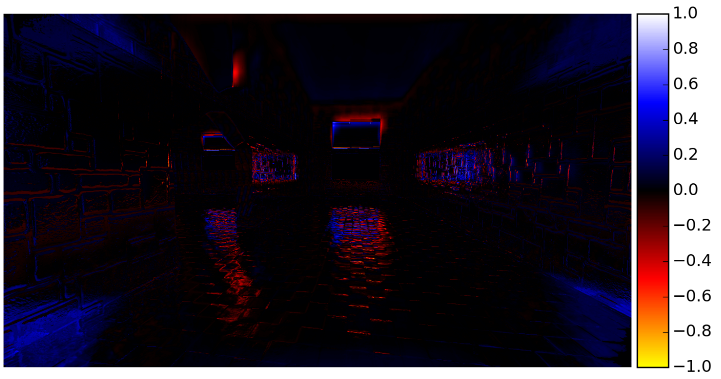
(b) SSLR rendering time ≈ 1.30 ms

Figure 5.4: Looking down to the floor at two cylinders, reflecting surrounding tiles (960 × 540 resolution).

In this image comparison (see Figure 5.5) we see two major advantages of SSCT over SSLR: In the first place we have a better precision, which is visible on the right wall (similar to Figure 5.3). Secondly the performance is clearly better for larger resolutions (here, a 1080p Full HD resolution). However, the performance partially depends on the scene setup and camera angle. But for this image the setup is advantageous for SSCT because the Hi-Z ray traversal can converge very fast. Nevertheless some visual errors are unavoidable in such screen space effects, even in SSCT like the small white spot shows, which is visible on the reflected passage's frame on the upper left ceiling.


(a) SSCT rendering time ≈ 6.83 ms

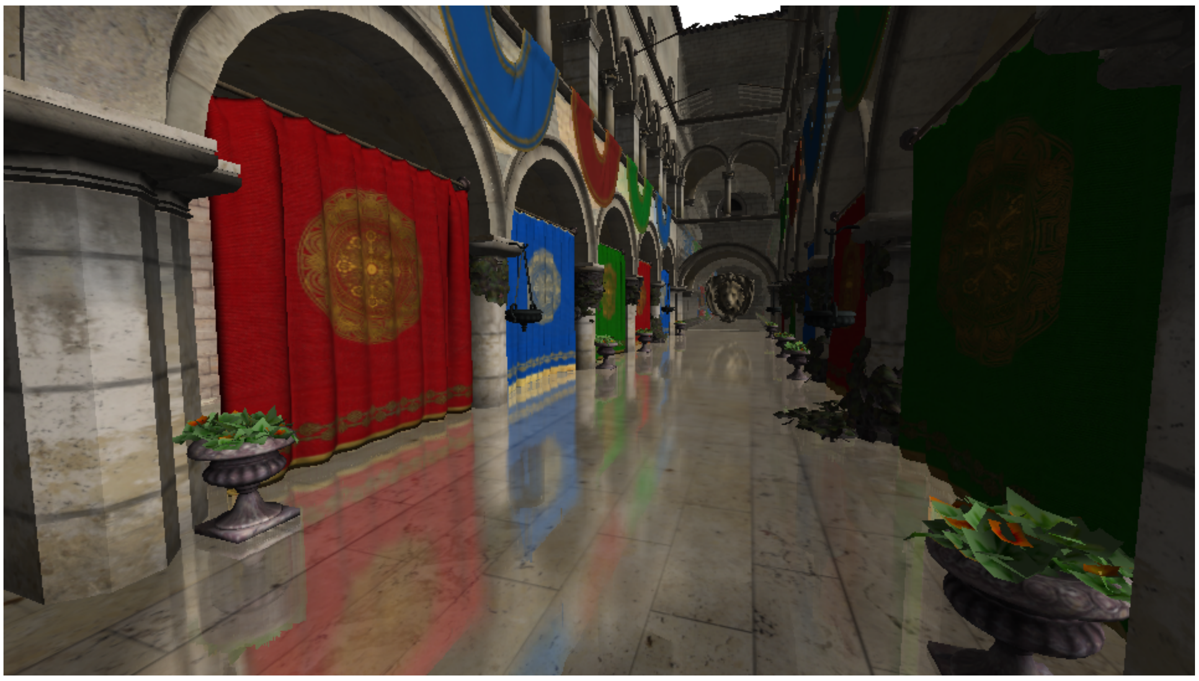

(b) SSLR rendering time ≈ 9.08 ms



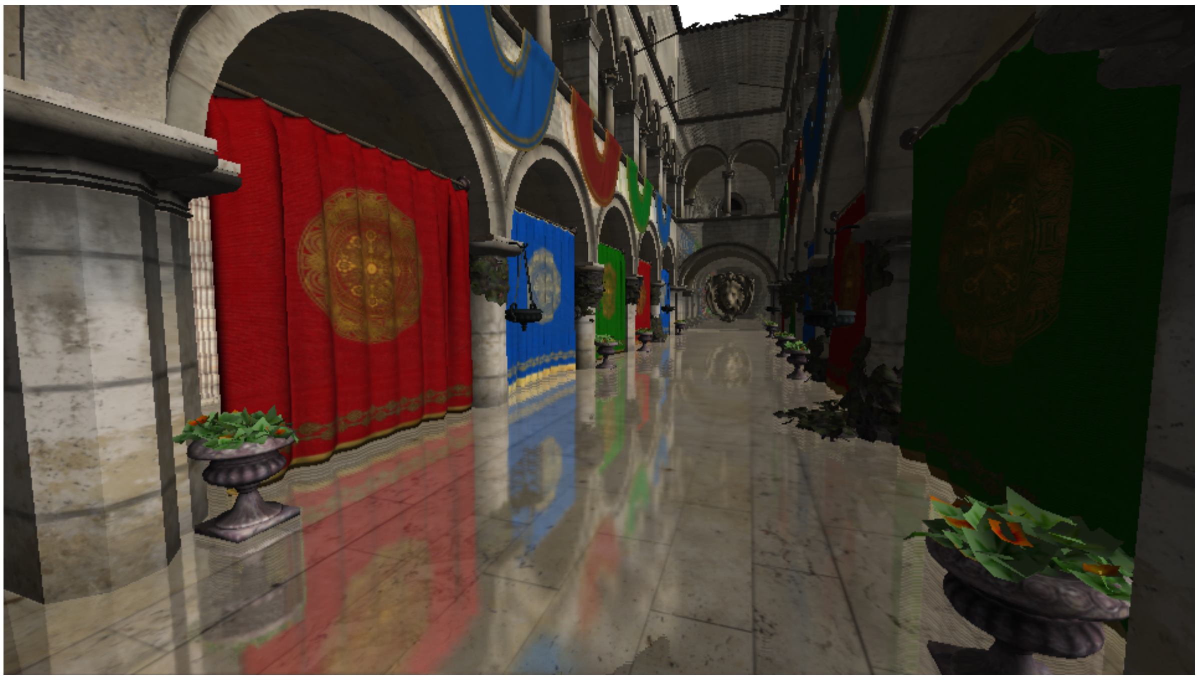
(c) Differential image

Figure 5.5: Looking to a window which is reflected in its surrounding (1920 × 1080 resolution).

In the following image comparison (see Figure 5.6) we see the *Sponza Atrium* [Dabrovic, 2002], which is a frequently used model for visual effect experiments and academic work. The scene itself is rendered neither with shadow mapping nor complex BRDF models. Only a single directional light source is embedded and only the stone floor reflects indirect light.



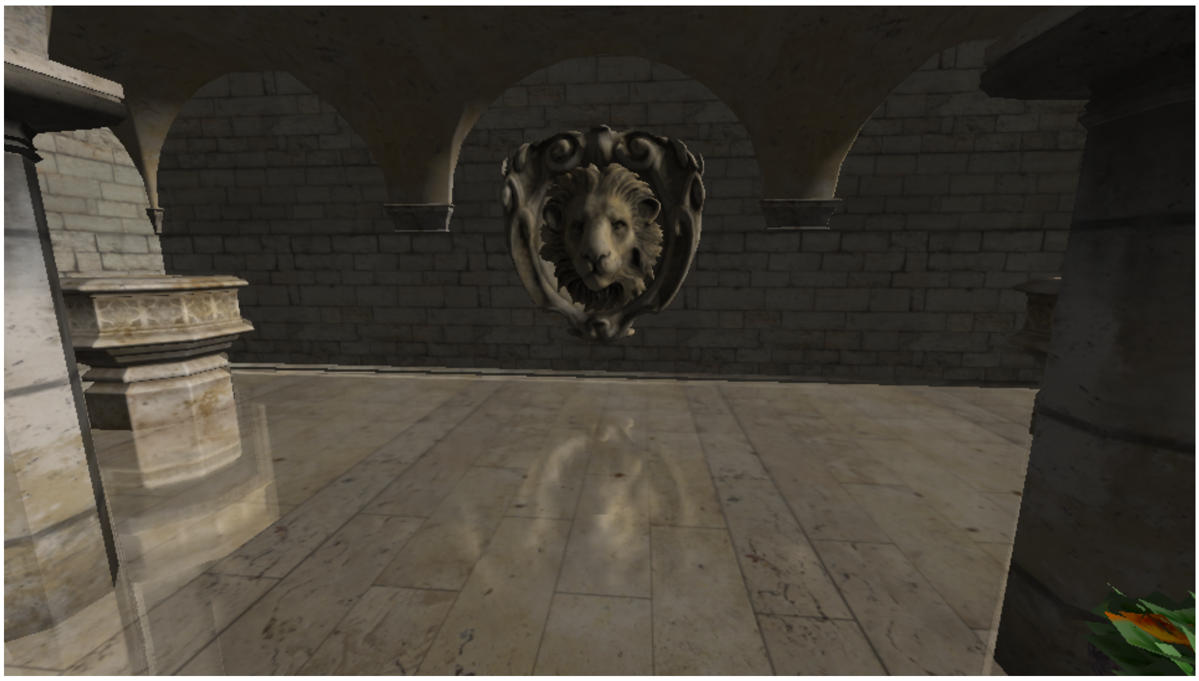
(a) SSCT rendering time ≈ 1.20 ms



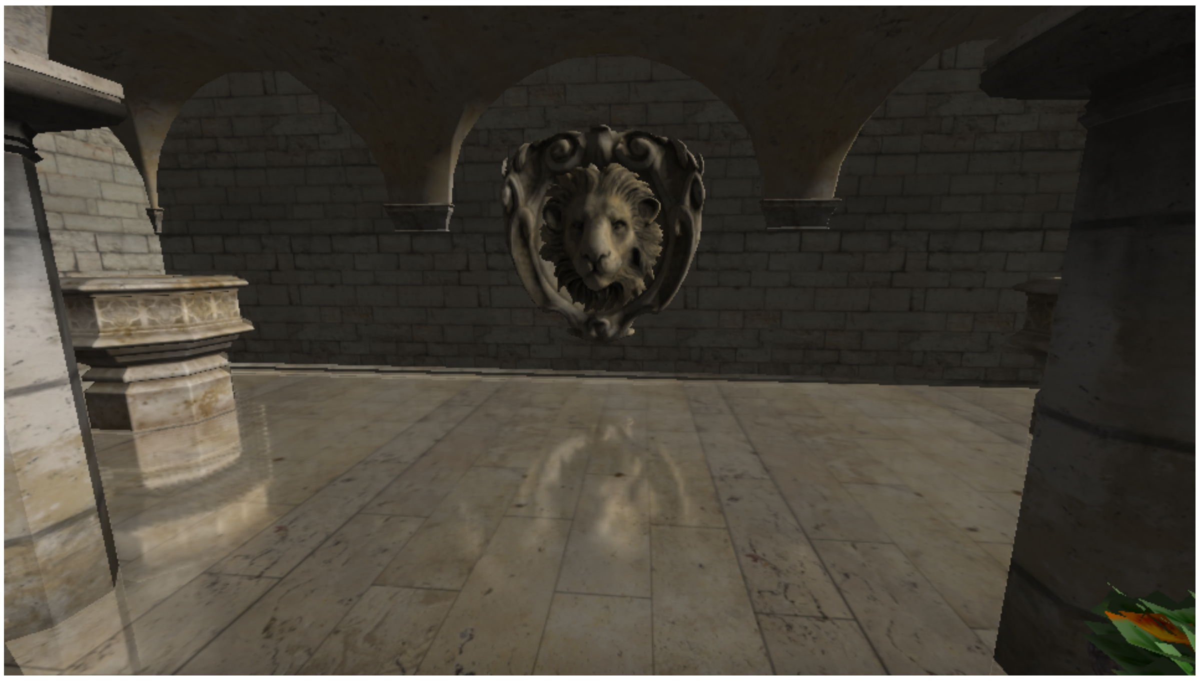
(b) SSLR rendering time ≈ 1.20 ms

Figure 5.6: Glossy reflections are cleary visible on the floor of the *Sponza Atrium* (960 × 540 resolution).

The lion head of the *Sponza Atrium* is reflected on the floor in this image comparison (see Figure 5.7).



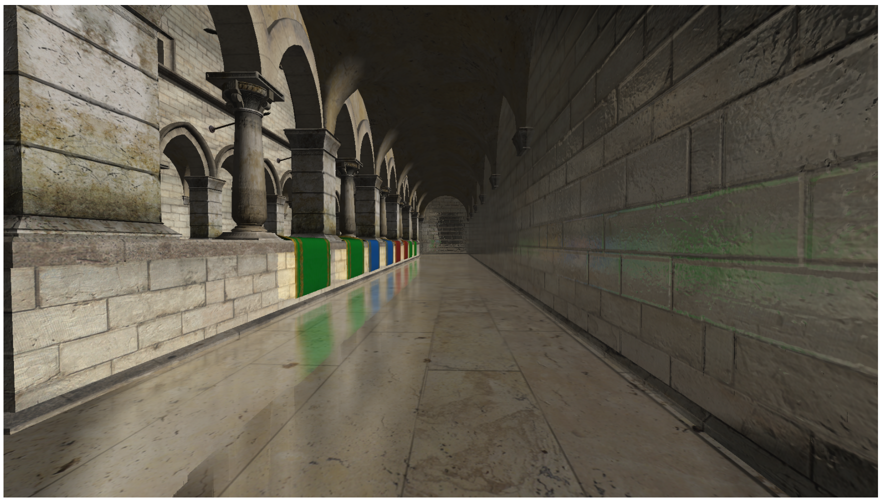
(a) SSCT rendering time ≈ 0.74 ms

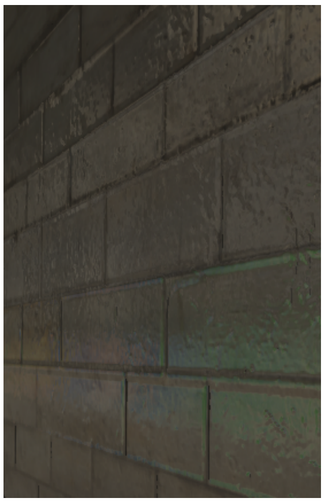


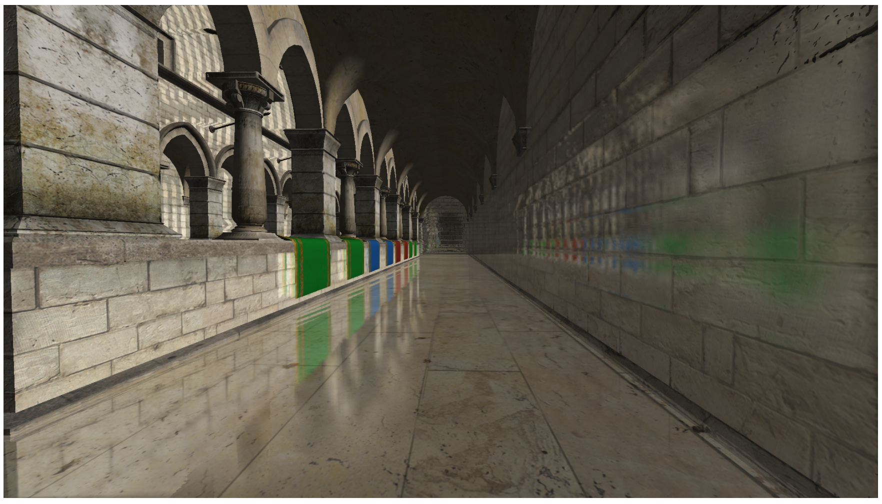
(b) SSLR rendering time ≈ 0.83 ms

Figure 5.7: The lion head of the *Sponza Atrium* is reflected on a rough surface (960 × 540 resolution).

In the following image comparison (see Figure 5.8) glossy reflections are visible on the floor and the walls of the *Sponza Atrium*. Here, SSCT produces much better results than SSLR for long ray traversals. Artifacts in SSLR are visible on the walls, due to a constant step size in the ray marching.


(a) SSCT rendering time ≈ 5.30 ms


(b) Enlarged image detail


(c) SSLR rendering time ≈ 5.19 ms

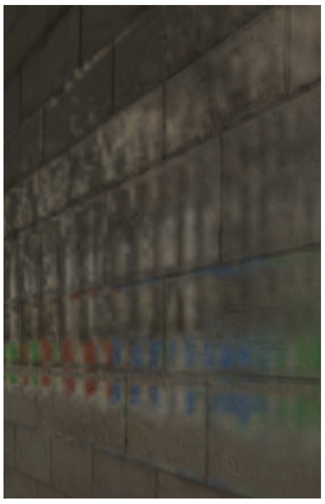
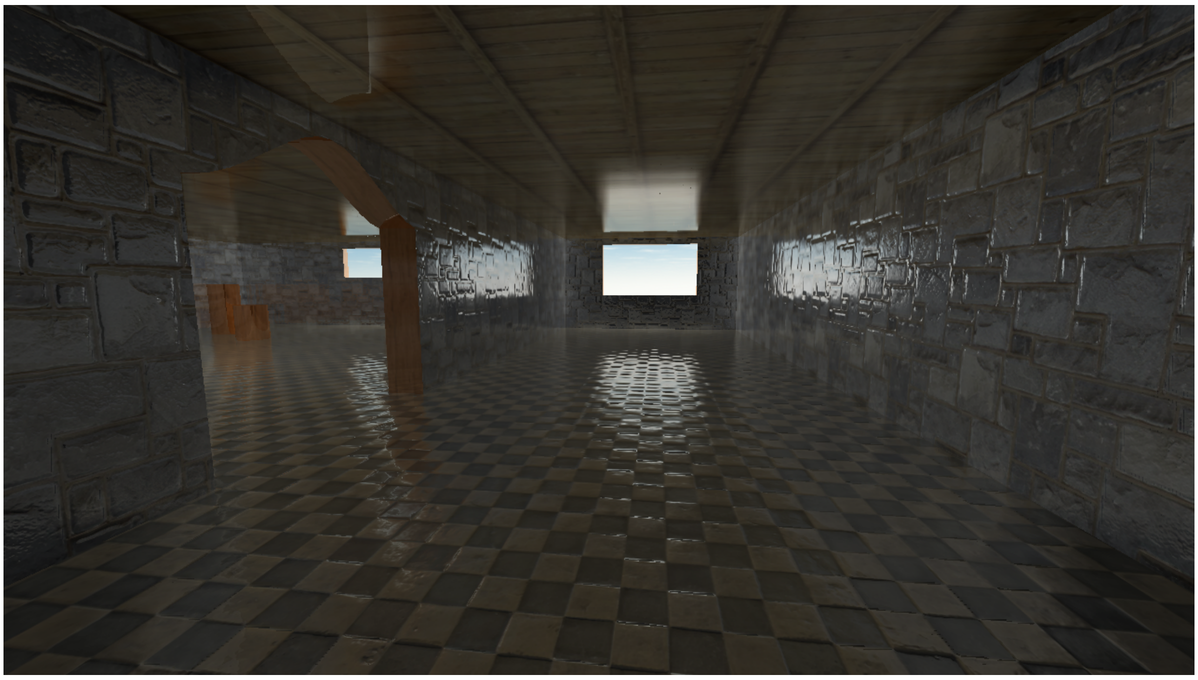

(d) Enlarged image detail

Figure 5.8: Glossy reflections are visible on the floor and the walls of the *Sponza Atrium* (1920 × 1080 resolution).

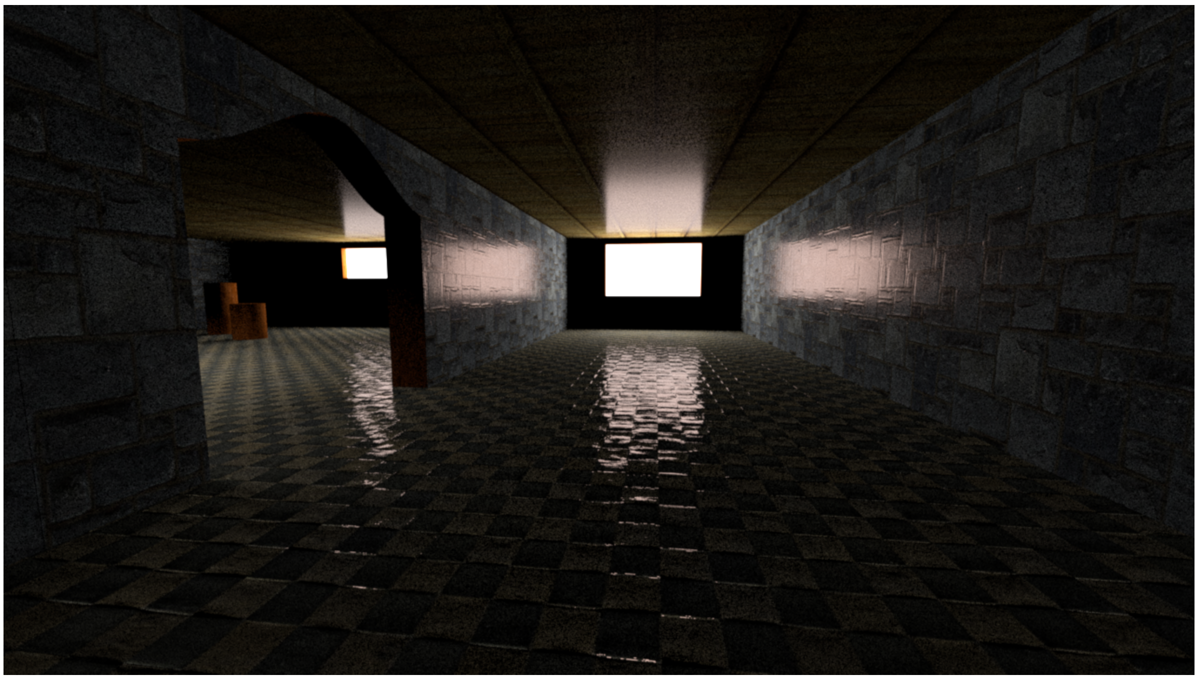
For this image there is also a table with performance results for various resolutions and roughness factors. The roughness factor is also used for the normal deviation, to increase the rough appearance.

Resolution	Roughness	SSCT Performance	HZCT Performance	SSLR Performance
640 × 480	0.0	0.76 ms	0.69 ms	1.14 ms
640 × 480	0.1	0.81 ms	0.74 ms	0.68 ms
800 × 600	0.0	1.12 ms	1.02 ms	1.69 ms
800 × 600	0.1	1.17 ms	1.10 ms	0.99 ms
960 × 540	0.0	1.50 ms	1.10 ms	1.86 ms
960 × 540	0.1	1.54 ms	1.19 ms	1.09 ms
1280 × 768	0.0	2.45 ms	1.92 ms	3.31 ms
1280 × 768	0.1	2.66 ms	2.07 ms	1.95 ms
1920 × 1080	0.0	4.92 ms	4.07 ms	6.09 ms
1920 × 1080	0.1	5.30 ms	4.49 ms	4.02 ms

This is the final image comparison (see Figure 5.9) which shows the difference between SSCT and the ground truth. Because there is no directional light in the Mitsuba scene, or any other ambient light, the shading in the SSCT scene has been dimmed. The rendering times are drastically different. This is the major benefit of this real-time approach over accurate rendering.



(a) SSCT rendering time $\approx$ 2.87 ms



(b) Mitsuba rendering time $\approx$ 04:05 min

Figure 5.9: The corridor scene again for comparison with the ground truth (1280 × 720 resolution).

6 Conclusion

We have seen a novel method for local glossy reflections called SSCT. The implementation has been presented in detail and a comparison to other state of the art methods has been shown as well. Our method is based on HZCT and is augmented with a fallback for special cases.

The advantages of our method over other SSLR methods are, that the cone tracing produces more plausible looking glossy reflections and it can be clearly separated from the ray tracing process. Additionally the input parameters for SSCT are more correlated with the material configuration of a BRDF. This is due to the cone tracing, which is derived from multiple ray samples. That makes it much easiy for artists to create plausible effects in 3D scenes. In contrast, most SSLR implementations merely blur the entire ray trace color buffer, which is out of proportion to BRDF and material parameters. Moreover the cone tracing in SSCT considers the amount of cone intersection with the scene and takes several texture samples, while glossiness in SSLR is usually based on a single and unweighted texture sample from the blurred ray trace color buffer. We can, though, make use of ideas implemented in SSLR: because of the modular nature of SSCT, we can further enhance the image quality by using a mask buffer in the blur pass for the color buffer, as explained in section 4.3.2.

However, our method still lacks solutions for the *hidden geometry* problem and the *screen boundary limitations*. Only workarounds do exist to circumvent these restrictions. We can summarize, therefore, local reflections in pure screen space effects are still an unsolved area of indirect lighting. Nevertheless, in prepared scenes and in combination with other reflection techniques it can be very useful with satisfying frame rates.

Acknowledgment

I would like to thank William Pearce ("WFP" from www.gamedev.net) for his code examples and support in the topic HZCT from the book *GPU Pro 5*. It helped me a lot with implementing the code base. I would also like to thank Jean-Philippe Grenier ("jgrenier" from www.gamedev.net) for his explanations and illustrations of the Hi-Z cone tracing part, which helped me a lot to overcome incomplete parts from the book.

Bibliography

John Amanatides. Ray tracing with cones. *SIGGRAPH Comput. Graph.*, 18(3):129–135, January 1984. ISSN 0097-8930. doi: 10.1145/964965.808589. URL http://doi.acm.org/10.1145/964965.808589.

Rui Bastos and Wolfgang Stürzlinger. Forward mapped planar mirror reflections. Technical report, 1998. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.1370.

Blender-Foundation. Blender, 1994. http://www.blender.org/.

A. T. Campbell and Donals S. Fussell. An analytic approach to illumination with area light sources, August 1991. ftp://ftp.cs.utexas.edu/.snapshot/backup/pub/techreports/tr91-25.pdf.

R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24, January 1982. ISSN 0730-0301. doi: 10.1145/357290.357293. URL http://doi.acm.org/10.1145/357290.357293.

Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum (Proceedings of Pacific Graphics 2011)*, 30(7), September 2011. URL http://maverick.inria.fr/Publications/2011/CNSGE11b.

Marko Dabrovic. Sponza atrium, 2002. http://www.crytek.com/cryengine/cryengine3/downloads.

Philip Dutre, Kavita Bala, Philippe Bekaert, and Peter Shirley. *Advanced Global Illumination*. AK Peters Ltd, 2006. ISBN 1568813074.

Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 213–222, New York, NY, USA, 1984. ACM. ISBN 0-89791-138-5. doi: 10.1145/800031.808601. URL http://doi.acm.org/10.1145/800031.808601.

Mark Henne and Hal Hickel. The making of "toy story". In *Proceedings of the 41st IEEE International Computer Conference*, COMPCON '96, pages 463–, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7414-8. URL http://dl.acm.org/citation.cfm?id=792769.793667.

Lukas Hermanns and Tobias Alexander Franke. Screen space cone tracing for glossy reflections. In *ACM SIGGRAPH 2014 Posters*, SIGGRAPH '14, pages 102:1–102:1, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2958-3. doi: 10.1145/2614217.2614274. URL http://doi.acm.org/10.1145/2614217.2614274.

IEEE. *IEEE standard for binary floating-point arithmetic*. Institute of Electrical and Electronics Engineers, New York, 1985. Note: Standard 754–1985.

Wenzel Jakob. Mitsuba renderer, 2010. http://www.mitsuba-renderer.org.

Mattias Johnsson. Approximating ray traced reflections using screen-space data, April 2012. http://publications.lib.chalmers.se/records/fulltext/193772/193772.pdf.

James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM. ISBN 0-89791-196-2. doi: 10.1145/15922.15902. URL http://doi.acm.org/10.1145/15922.15902.

Jan Kautz and Michael D. McCool. Approximation of glossy reflection with prefiltered environment maps. In *In Graphics Interface*, pages 119–126, 2000.

Khronos-Group. Opengl shading language, 2004. https://www.opengl.org/documentation/glsl/.

Sébastien Lagarde and Zanuttini Antoine. Local image-based lighting with parallax-corrected cubemaps. In *ACM SIGGRAPH 2012 Talks*, SIGGRAPH '12, pages 36:1–36:1, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1683-5. doi: 10.1145/2343045.2343094. URL http://doi.acm.org/10.1145/2343045.2343094.

Jason Lawrence. Importance sampling of the phong reflectance model, 2002. www.cs.virginia.edu/~jdl/importance.doc.

Kok-Lim Low. Perspective-correct interpolation, 2002. http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.3.211.

Morgan McGuire and Michael Mara. Efficient GPU screen-space ray tracing. *Journal of Computer Graphics Techniques (JCGT)*, 3(4):73–85, December 2014. ISSN 2331-7418. http://jcgt.org/published/0003/04/04/.

POV-Team. Persistence of vision pty. ltd., williamstown, victoria, australia, 2004. http://www.povray.org/.

Shadertoy. Area lights - shadertoy, October 2013. https://www.shadertoy.com/view/ldfGWs.

Peter Sikachev and Nicolas Longchamps. Reflection system in thief, August 2014. http://advances.realtimerendering.com/s2014/#_REFLECTION_SYSTEM_IN.

John A. Tsakok. Faster incoherent rays: Multi-bvh ray stream tracing. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 151–158, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-603-8. doi: 10.1145/1572769.1572793. URL http://doi.acm.org/10.1145/1572769.1572793.

Yasin Uludag. Hi-z screen-space cone-traced reflections. In Wolfgang Engel, editor, *GPU Pro 5*, pages 149–192. CRC Press, 2014.

Steve Upstill. *RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989. ISBN 0201508680.

Michal Valient. Reflections and volumetrics of killzone shadow fall, August 2014. http://advances.realtimerendering.com/s2014/#_REFLECTIONS_AND_VOLUMETRICS.

Dietger van Antwerpen. Improving simd efficiency for parallel monte carlo light transport on the gpu. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 41–50, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0896-0. doi: 10.1145/2018323.2018330. URL http://doi.acm.org/10.1145/2018323.2018330.

Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. Embree: A kernel framework for efficient cpu ray tracing. *ACM Trans. Graph.*, 33(4):143:1–143:8, July 2014. ISSN 0730-0301. doi: 10.1145/2601097.2601199. URL http://doi.acm.org/10.1145/2601097.2601199.

Daniel Wright. Image based reflections. https://udn.epicgames.com/Three/ImageBasedReflections.html.

Acronyms

AAL	Analytical Area Light
AR	Augmented Realtiy
BRDF	Bidirectional Reflectance Distribution Function
BSP	Binary Space Partitioning
BVH	Bounding Volume Hierarchy
CAD	Computer Aided Design
CGI	Computer Generated Imagery
G-Buffer	Geometry Buffer
GI	Global Illumination
GLSL	OpenGL Shading Language
HLSL	DirectX High Level Shading Language
HZCT	Hi-Z Cone Tracing
IBR	Image Based Reflections
LPV	Light Propagation Volume
MIP	Multum In Parvo
NPOT	None Power Of Two
PDF	Probability Density Function
POT	Power Of Two
SIMD	Single Instruction Multiple Data
SSCT	Screen Space Cone Tracing
SSLR	Screen Space Local Reflections
UE3	Unreal Engine 3
VCT	Voxel Cone Tracing
VPL	Virtual Point Light