Acknowledgements

I would like to thank the following people.

- Patrick Cozzi for giving me the support and freedom to explore my research.
- Adam Mally and Daniel McCann, for reviewing and helping to fix grammar errors.
- My thesis committee and supervisor, Stephen Lane and Norman Badler, for giving me the freedom to take this project in any direction.
- Adam Cichocki, the author of Optimized Pixel-projected reflections, for helping to implement his technique.

Figure 1: Optimized pixel-projected reflections for planar reflectors (0.779 ms on a GTX1060 in full HD resolution 1920 x 1080).

Abstract

Achieving dynamic global illumination in a real-time environment is a difficult issue. To obtain dynamic reflections, many screen-space reflections techniques using ray marching have been studied, but they are expensive to use for real-time rendering. In this thesis, we surveyed previous screen space reflections techniques. After that, it covers how to implement Adam Cichocki's new screen-space reflections technique using ray tracing (in GLSL with Vulkan API) and improve.

1. Introduction

In this thesis, we introduce the Vulkan implementation of optimized Pixel-projected reflections for planar reflectors presented in SIGGRAPH 2017 [Cichocki17]. Furthermore, we describe how to handle this technique with physically-based rendering environment.

Traditionally, screen space reflections use ray marching. The downside is that the performance of current hardware is not sufficient to apply very small ray steps in real time. If we increase the step size of ray marching the performance improves, but the quality of the result is worse. To solve this problem, many studies have been carried out to optimize ray marching or apply post-process effects to the results of screen space reflections to reduce artifacts. Nevertheless, the core issues caused by ray marching still exist. For example, unless we use a very tiny step size per ray iteration, it is difficult to get the exactly correct color reflected in the screen space image. In other words, ray-marching in real-time is too slow.

To solve this, [Cichocki17] describes a method using ray tracing instead of ray-marching. Not only can ray tracing can find the intersection between rays and objects in a single operation, but there is also no possibility of inaccurate intersections returned by ray marching. Therefore, we can get more accurate results with lower cost. However, using ray tracing introduces a rather large limitation: we need to test if the reflector surfaces intersect with our ray. In particular, it is expensive to install reflectors that exactly match the surfaces of moving objects in real time. Thus, the idea devised by Adam Cichocki deals with reflections on basic planar surfaces.

While this is a significant limitation, there are many common cases where simple flat reflections are useful such as mirror reflection, serene lake water, indoor floors, and so on.

The original work deals only with perfect specular reflections. But, in this thesis, we extend it to incorporate microfacet roughness so ray-traced reflections can be naturally integrated into a physically-based rendering system. Furthermore, we explain how to implement this idea in the GLSL language with Vulkan's API because the original algorithm used HLSL in DirectX.

2. Previous Work

This chapter is the survey section of this thesis.

Screen space reflections began with attempts to bring global lighting (GI) from the original scene image in real-time. The most basic algorithm is to perform ray marching from each pixel in the screen space along the direction of individual reflection vector. Then, obtain the color when the end point from ray marching intersects with an object in the scene. In other words, iterate ray marching until the ray intersects against the depth buffer. However, in order to get an accurate result, very small ray-marching step size must be used to avoid banding artifacts. Thus, with current hardware capabilities, it is difficult to use it in real time.

There have been many attempts to solve this problem. The most frequent and essential approach is optimization of ray marching methods. In [Wronski14], [Stachowiak15] and [Giacaone16], screen space is divided into smaller tiles and ray marching is performed in a cheap and imprecise pass before a final detail pass. [Valient14] used temporal re-projection which renders each update in a half resolution buffer then upscales it to full resolution. [Uludag14] used ray marching with pixel unit step size from a variable mip level depth buffer until the ray intersects to prevent the ray from skipping over tiny geometry while mostly marching at a cheaper resolution.

After physically-based microfacet rendering became the essential lighting model for real-time rendering, there were efforts to apply it to screen space reflections. For a plausible BRDF match, [Valient14] used mip-mapped scene image to get blurred color

corresponding to the reflective surface roughness. [Stachowiak15] used BRDF importance sampling and reused the neighbor pixel information to blur the result.

2.1. Ray Tracing

Ray tracing is a rendering technique that traces the ray's path from the pixels in the imaginary screen space image plane to produce the result image. This technique is mainly used to generate a high degree of visual realism, but it also requires a high cost. Therefore, it is used as a visual effect in television, movie or 3D animation for still images rather than real-time rendering.

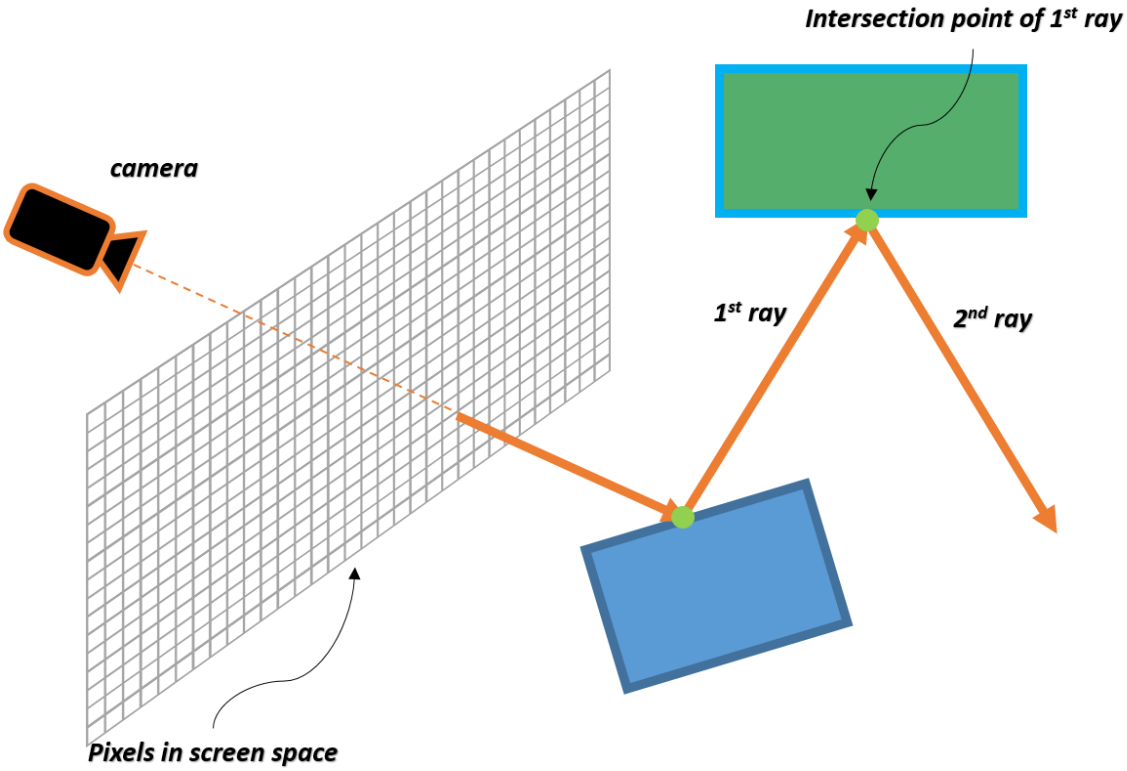

Figure 2: Illustration of ray tracing.

2.1.1. Ray Tracing Algorithm

The first ray tracing algorithm used for rendering was presented by Arthur Appel in 1968 [Appel68]. This thesis is the first mention of tracing the rays (Fig. 2) shot from an eye (or camera) to find the closest objects. Later, there are many methods to develop ray tracing, but this core idea does not change. If an object is found, a reflection vector can be calculated using the surface normal vector at the intersection, which can then be traced to find reflected color. This can continue for an arbitrary number of light bounces. To compute the shadow, at the point where the pixel in screen space, shoot a ray, which is called shadow feeler, again to the light source and check for its visibility. This will result in a hard shadow, but using multiple samples will result in a soft shadow based on the averaged result.

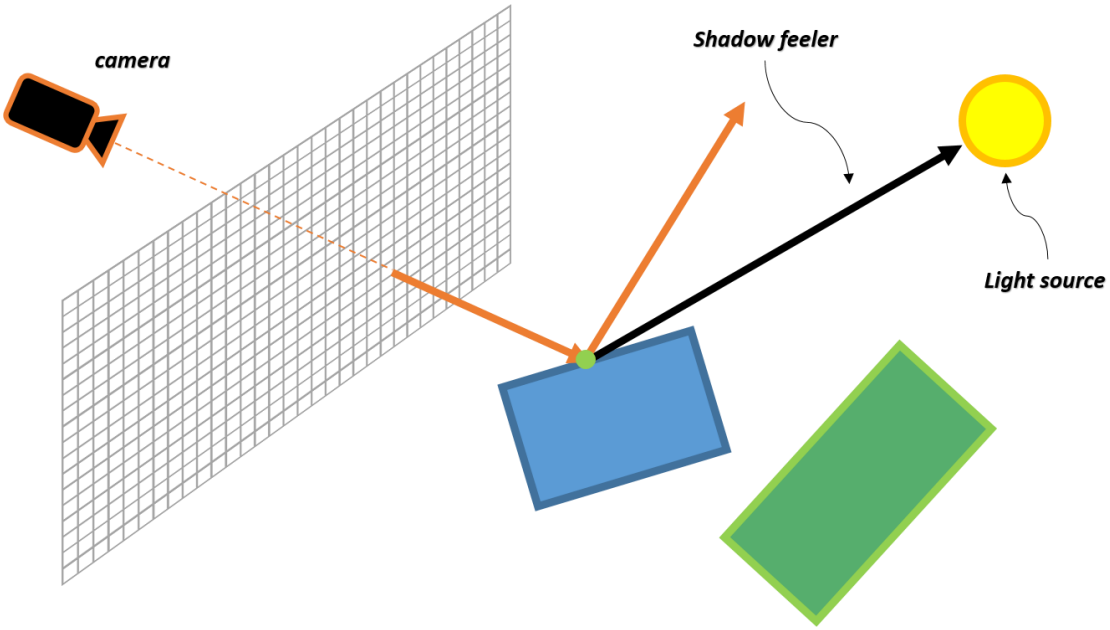

Figure 3: To compute the shadow, shadow feeler is shot from the intersection point of 1st ray. In this image, shadow feeler reaches to the light source, directly. Therefore, the point is not shadow area.

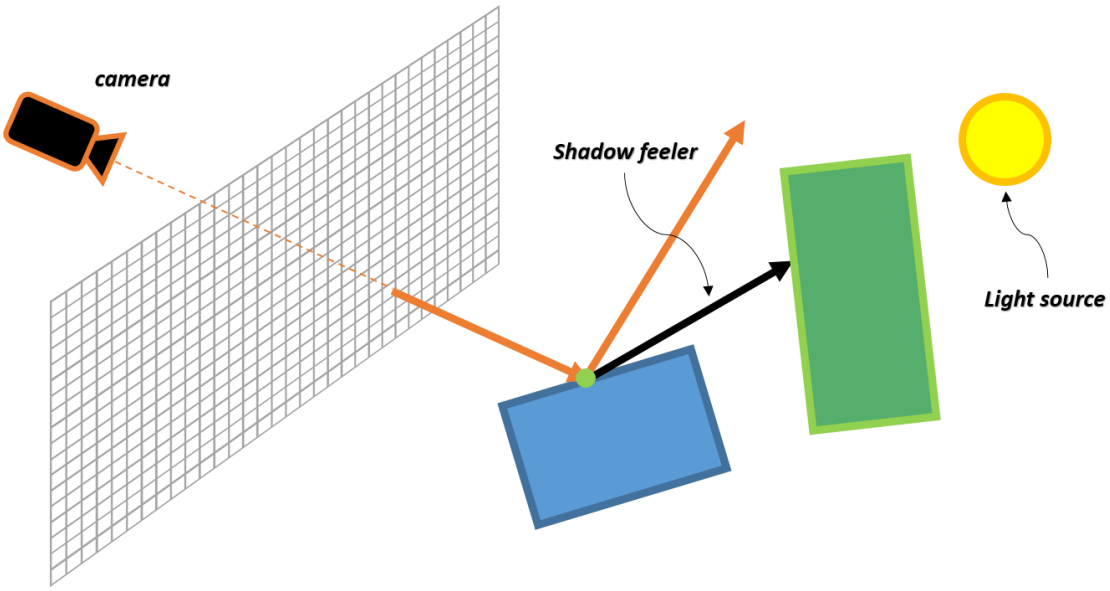

Figure 4: In this image, shadow feeler can't reach to the light source (It is blocked by the green object). Then, the point is shadow area.

2.1.2 Optimization

As mentioned before, ray tracing requires high cost. Below introduces popular ways to optimize it.

Using primitive shapes

It is not easy to determine whether a complex object intersects a ray or not. This is because the object must be able to be represented by a single mathematical function, the more complex the object, the more difficult to create its function, and the complexity of the calculation itself to determine whether it intersects with the ray also increase. For this reason, primitive shapes (triangle, plane, cube, sphere, etc) are usually used for fast calculations. The intersection of primitive shapes and rays requires only a simple

calculation. In particular, most 3D objects consist of triangles or planes (quads), so they can be applied without exception.

Bounding volumes

If the direction of the ray is not pointing at the target object at all, we can prevent to traverse to find the intersection of the ray with the object in advance. To do this, perform ray tracing first on the bounding volume (Typically, a simple primitive, such as a cube or a circle, is used) covering the entire object. If there is an intersection point, perform the ray tracing again to find the point of intersection with the object, and if there is no intersection point, skip to the next object.

Acceleration structures

Complex objects consist of many triangles or quads, and it is inefficient to traverse them all. To improve this, divide the object into a hierarchical spatial structure first, then test whether its child nodes intersect with a ray from the root node. If it intersects, the child node becomes the parent node and test against its child nodes. This process is repeated until find the closest intersection point. The most well-known methods are BVH (Bounding volume hierarchy) and KD-Tree. However, the cost for creating this hierarchy in advance is needed.

2.2. Ray Marching

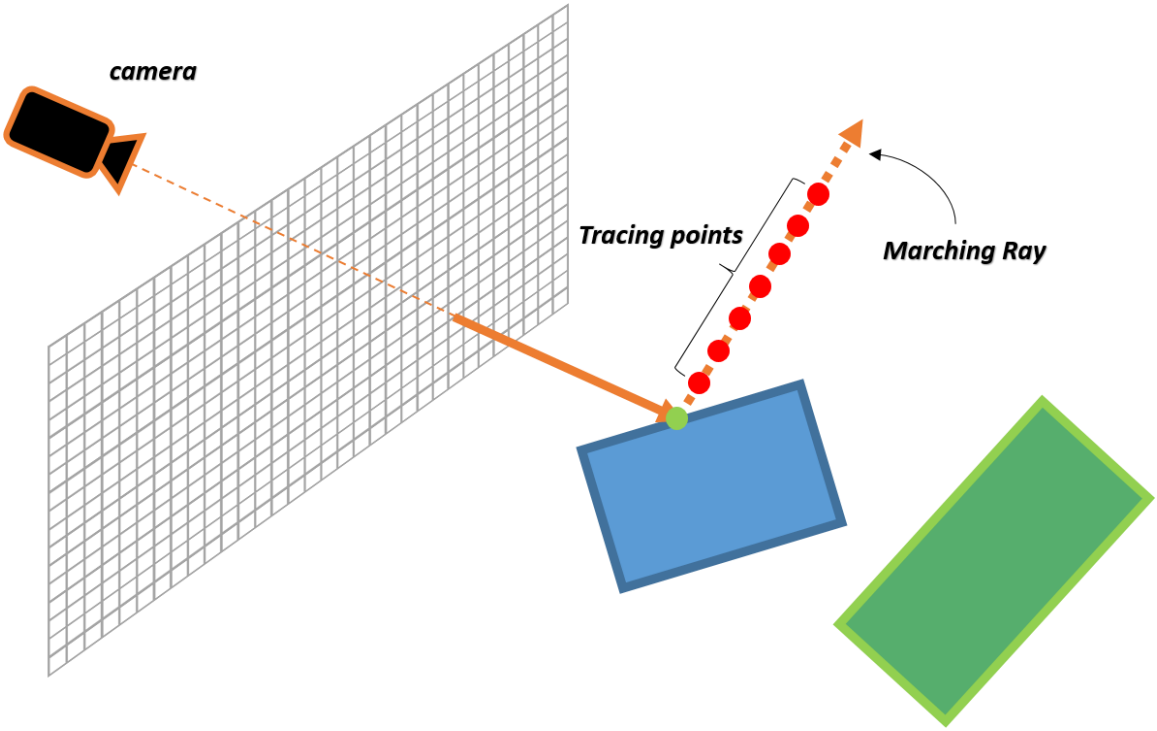

Figure 5: Illustration of ray marching.

Ray marching is a technique similar to ray tracing. Intersection tests in ray tracing find a location on an explicit surface at the point of intersection, but intersection tests on implicit surfaces only test whether a point is on a surface or not. To find the surface point, many points along the ray must be tested. In the case of screen space reflections, the target implicit surface is the camera's depth buffer.

From the reflective surface, march along the reflected ray, and project each march point to screen space and compare its depth with depth buffer. If the depth is larger than depth buffer, we judge that the ray intersected the object represented at that part of the depth buffer.

Step size is the most important factor, and is difficult to calibrate: if the size is too small, finding the intersection is too slow, but if it is too large, it becomes possible to overshoot the true point of intersection on the geometry (Step size should not be always uniform. For example, [McGuire14] used adaptive step size with binary search around last step segment). However, the reason why ray marching is useful for reflection is that ray tracing is too expensive in real-time scenes without adequate acceleration structures, and these structures are too difficult to build and rebuild every frame for moving objects. For this reason, previous methods of screen space reflections have evolved based on ray marching.

2.3. Screen Space Reflections

The following methods are based on the idea of using ray marching to obtain color information by ray marching against the screen's depth buffer.

2.3.1. Brute Force Screen Space Reflections

This is the most basic screen space method. First, calculate the reflection vector using the normal and view vectors in the screen space for each pixel. Next, advance the tracing point by a certain step size along the direction of the reflection vector. Each time the tracing point moves forward, compare depth value obtained by projecting the current tracing point into the screen space with depth buffer. If the depth value of the tracing point is larger than the depth buffer, stop ray marching and obtain the scene image pixel value of the current point.

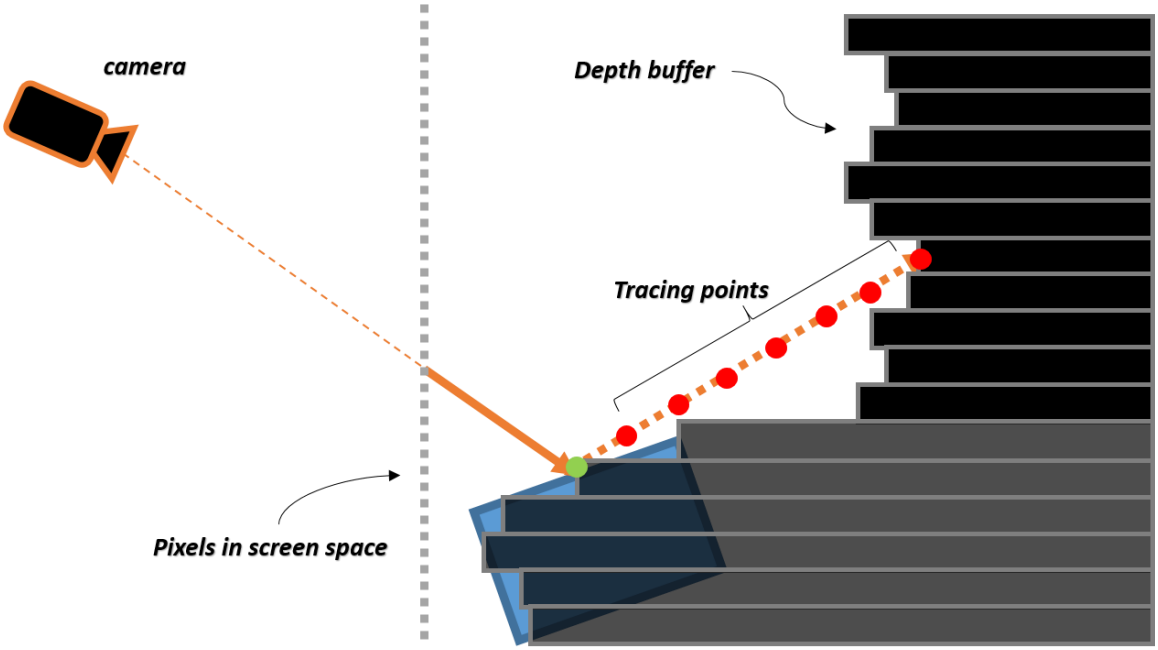

Figure 6: Illustration of basic screen space reflections.

There are two notable pitfalls that must be accounted for. The first is that there can be objects that are much closer to the camera than the ray path, and the depth comparison test will falsely mark it as an intersection. In this case, we specify a certain threshold according to the step size, and if the difference value is larger than the threshold, the

result value is not used. Second, if the intersection point is not found and the projected tracing point is out of the screen space, the ray marching must be stopped.

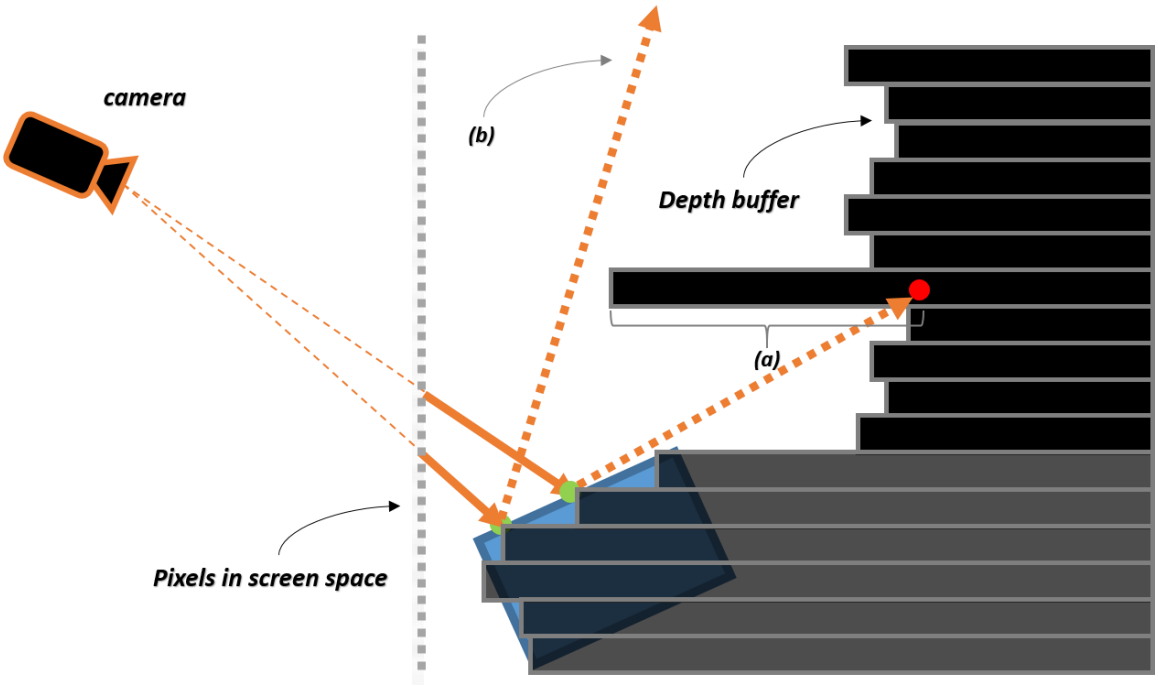

Figure 7: The cases when we discard the result. (a) If the difference between the depth value of the tracing point and depth buffer is too large, the result is not accurate anymore. (b) If the ray marching ray is out of the screen, it result also should be discarded.

Using a small step size gives the best screen space reflections quality, but it is hard to maintain 60 fps (16.666ms) in real time due to the current hardware performance. Using a downsampled frame buffer at 1/4 resolution is a typical solution. Note that when downsampling the depth buffer, the minimum value among the pixels filtered together should be used to prevent the ray from missing the closest intersection.

To use a larger step size with fewer artifacts, [Val14] uses simple linear interpolation between a hit point and the previous point with depth buffer values of the points.

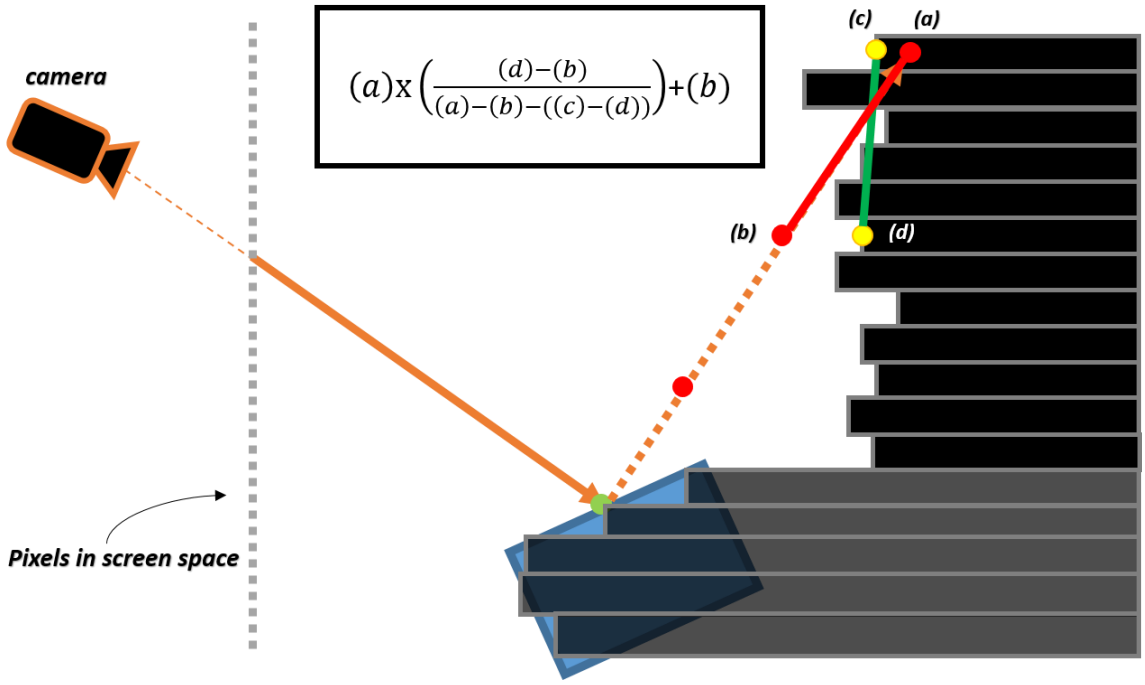

Figure 8: Depth linear interpolation between the last tracing point and previous tracing point. *(a)* : the depth of the last tracing point. *(b)* : the depth of previous tracing point. *(c)* : the value of depth buffer at the last tracing point. *(d)* : the value of depth buffer at previous tracing point. The limitation of this method is that it cannot be adjusted when the first step of ray marching hit an object because *(b)* and *(d)* are same.

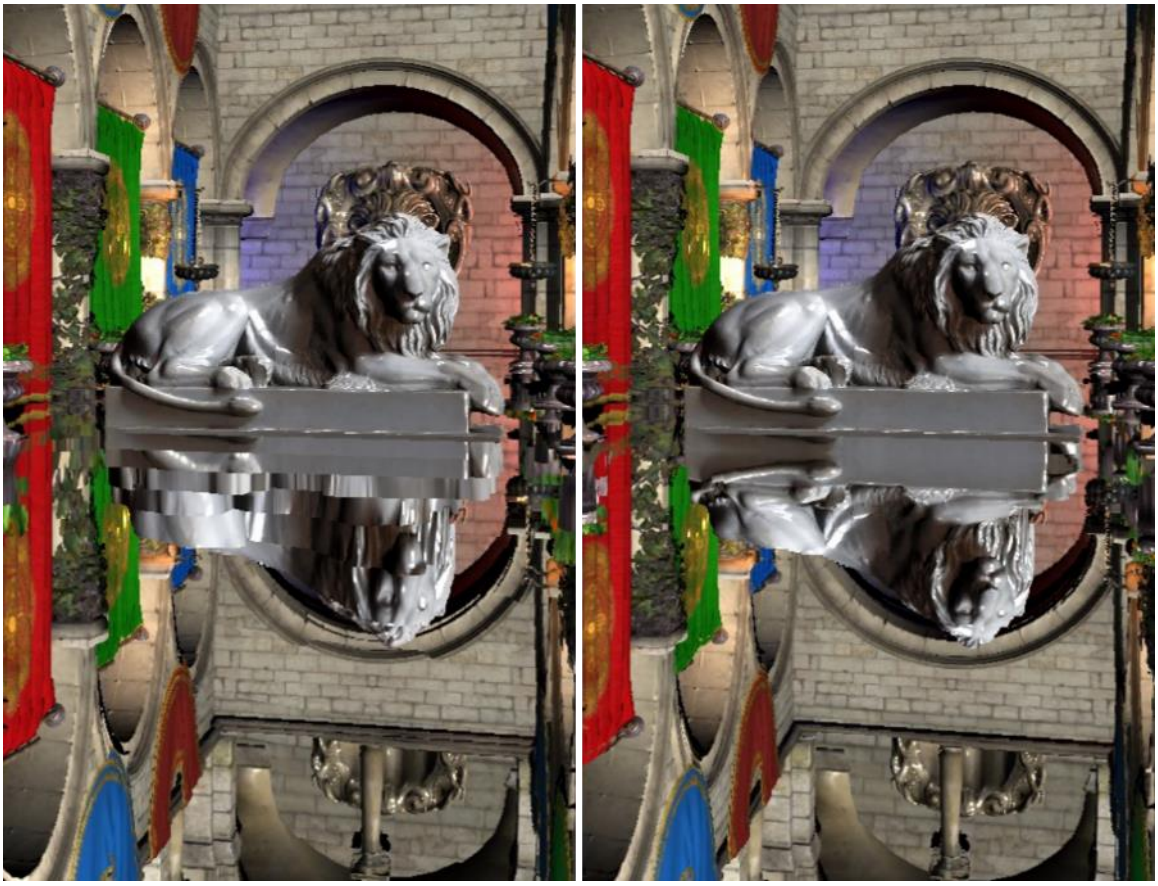

Figure 9: Comparison of the result without / with depth linear interpolation. Left: without depth linear interpolation. Right: with depth linear interpolation. Both images are using max iteration 128, step size 5.0.

2.3.2. Hierarchical Depth Screen Space Reflections

[Uludag14] devised pixel-based ray marching using a hierarchical depth buffer, also known as the Hi-Z buffer. As mentioned earlier, the Hi-Z buffer also creates a hierarchical structure by using a downsampling process that compresses the information of four neighboring pixels into one pixel. Therefore, the minimum value among the pixels filtered together should be written.

Figure 10: Illustration of 4 neighboring pixels at hierarchical depth buffer. Left: before downsampling Right: after downsampling (This figure based on [Uludag14])

After generating the Hi-Z buffer, ray marching starts on the first mip level. If there is no intersection, then increase the mip level try again at the target point. (Fig. 6-1) If the target point hit an object, decrease the mip level and do ray marching again at previous target point's position. Repeat until the target point hits an object against minimum level of Hi-Z buffer. (Fig. 6-2) In this way, we can prevent a ray from passing through tiny objects. However, it costs extra to create a depth buffer hierarchy for every frame.

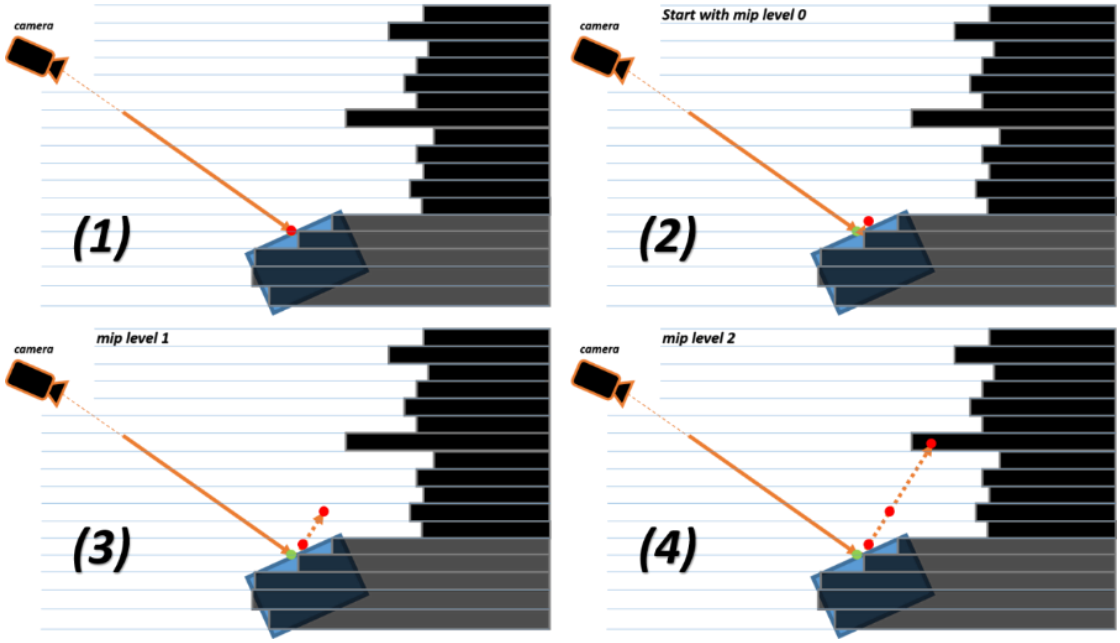

Figure 11: Ray marching is started with first mip level. If there is no intersection, increase the mip level and do ray marching again at the target point.

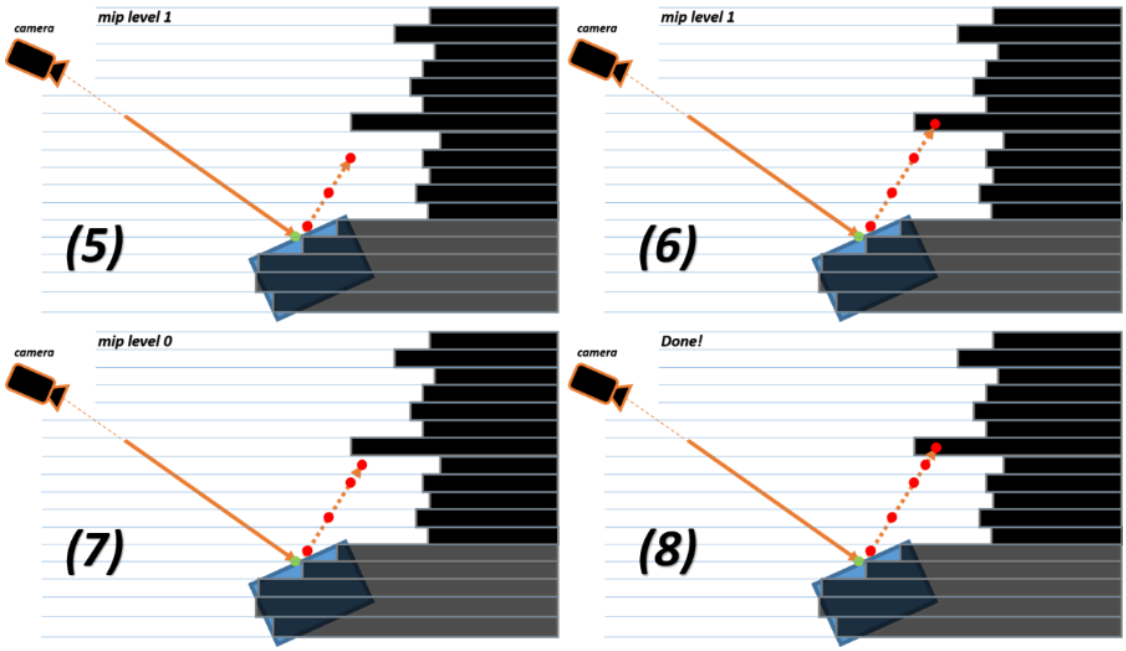

Figure 12: If the target point hit an object, decrease the mip level and do ray marching again at previous target point's position. Iterate these steps until our target point hit an abject against minimum level of Hi-Z buffer.

2.3.3. Tile Classification

For most scenes, a vast majority of pixels in screen space do not contribute to the final reflection result. With this idea, [Giacaone16] used tile classification to discard non-contributing pixels. We can divide the screen into square tiles. Then, for each tile, create some small number of test rays with sparse ray distribution [Wronski14], which uses non-uniform jittered samples to prevent artifacts caused by using uniform sampling. If none of the test rays hit, we can discard the non-contributing tile for saving cost. Otherwise, march on each pixel in the tile.

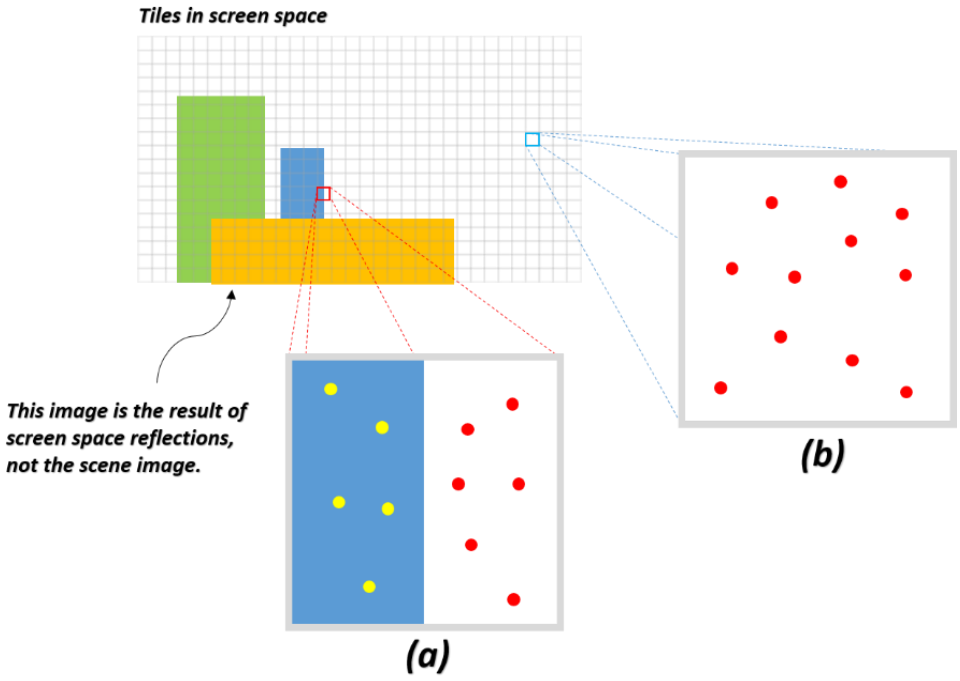

Figure 13: Illustration of tile classification. In tile (a), half of samples are contributing to the result. Thus, it is worth to do ray marching at whole pixels in the tile. However, in tile (b), there is only non-contributing samples. Therefore, we can discard this tile.

3. Vulkan

Vulkan is a modern graphics API designed by the Khronos Group. Khronos provides full Vulkan specification. To start developing applications using the Vulkan API, one downloads the Vulkan SDK from LunarG Developers. Vulkan is an explicit graphics API that gives programmers more opportunities to optimize their rendering pipelines. Vulkan API is designed for next-generation graphics hardware, multi-threaded rendering and VR/AR.

3.1. Overview

This section describes the concepts for creating a minimal rendering framework with Vulkan API. The explanations below are based on [Lapinski17].

Instance

A **`VkInstance`** is an object that gathers the state of an application. It encloses information such as an application name, name and version of an engine used to create an application, and enables instance-level extensions and layers. Through the Instance, we can also enumerate available physical devices and create logical devices on which typical operations such as image creation or drawing are performed.

Physical Devices and Logical Devices

Almost all the work in Vulkan is performed on logical devices such as creating resources, managing memory, recording command buffers and submitting commands. Logical devices represent physical devices for which a set of features and extensions are enabled. To create a logical device, we need to select one of the physical devices **`VkPhysicalDevice`** available on a given hardware platform.

Presentation Surface and Swapchain

A presentation surface represents an application's window. It allows us to acquire the window's parameters, such as dimensions, supported color formats, required number of images, or presentation modes. A swapchain is used to display images on the screen. It is an array of images which can be acquired by the application and then presented in our application's window. Each image has the same defined set of properties such as a size, a format, and usage.

Image Views and Framebuffers

Images are rarely used directly in Vulkan commands. Framebuffers and shaders access images through image views. Image views define a selected part of an image's memory and specify additional information needed to properly read an image's data. Framebuffers are used along with render passes. They specify what image resources should be used for corresponding attachments defined in a render pass. They also define the size of a renderable area.

Render Passes

Rendering can only be performed inside render passes. When we also want to perform other operations such as image post processing or preparing geometry and light pre-pass data, we need to order these operations into sub-passes. For this, we specify descriptions of all the required attachments, all sub-passes into which operations are grouped, and the necessary dependencies between those operations.

Graphics Pipeline

A graphics pipeline is the object that allows us to draw anything on screen. It controls how the graphics hardware performs all the drawing-related operations, which transform vertices provided by the application into fragments appearing on screen.

Compute Pipeline

A compute pipeline is the second type of pipeline available in the Vulkan API. It is used for dispatching compute shaders, which can perform any mathematical operations. And

as the compute pipeline is much simpler than the graphics pipeline, we create it by providing far fewer parameters.

Command Pools and Command Buffers

Command pools are objects from which command buffers acquire their memory. Memory itself is allocated implicitly and dynamically, but without it command buffers would not have any storage space to hold the recorded commands. Command buffers are used to store commands that are later submitted to queues, where they are executed and processed by hardware to give us results.

Semaphores, Fences and Queues

Before we can submit commands and utilize the device's processing power, we need to know how to synchronize operations. Semaphores are one of the primitives used for synchronization. They allow us to coordinate operations submitted to queues, not only within one queue, but also between different queues in one logical device. Fences, similarly to other synchronization primitives, have only two states: signaled and un-signaled. They can be created in either or theses two states, but they are reset by the application. To signal a fence, we need to provide it during command buffer submission. Such a fence, similarly to semaphores, will become signaled as soon as all work submitted along with the fence is finished. Queues are used to submit commands to the device to allow specification of shader programs, kernels and the data used by the kernels.

3.2. Implementation

Our application is structured and pipeline as followed:

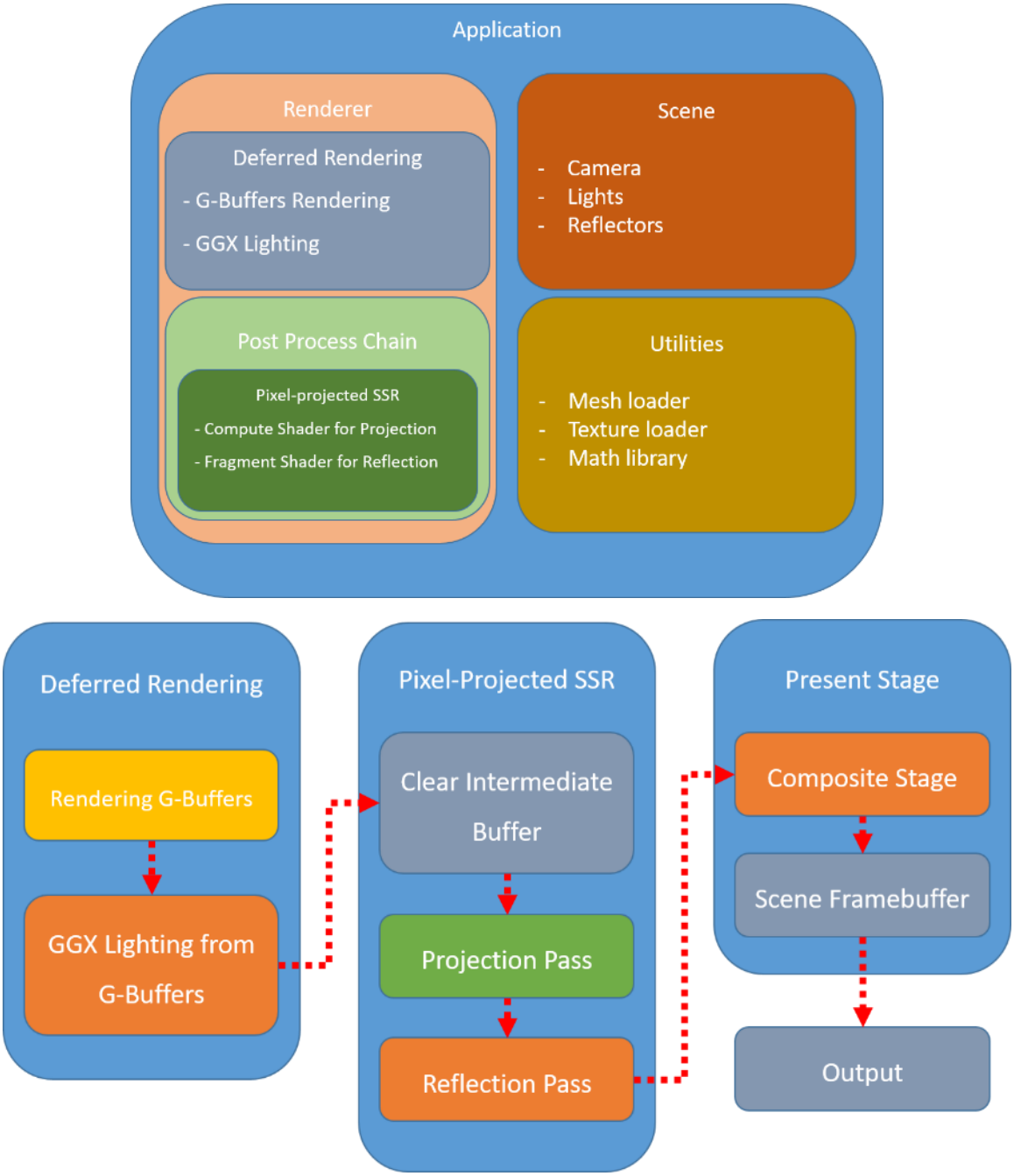

Figure 14: Illustration of the structure and pipeline of our application.

Our application uses deferred rendering because it is natural to get and reuse normal data in screen space for screen space reflections. We used 3 G-buffers (first one is for albedo and opacity, second one is for specular color and roughness, third one is for normal and metallic) of which format are VK_FORMAT_R8G8B8A8_UNORM, VK_FORMAT_R8G8B8A8_UNORM and VK_FORMAT_R32G32B32A32_SFLOAT. And, we used VK_FORMAT_D32_SFLOAT format for Depth buffer. For screen space rendering stages (Orange round rectangles in Fig.14), we use a single triangle to fill whole screen instead of the traditional quad, which consists of two triangles, for better performance according to [Akenine18]. In Pixel-Projected Reflections stage, we clear the Intermediate Buffer (described in 4.2) on CPU side (Gray round rectangles in Fig.14) . Then, in a compute shader (Green round rectangles in Fig.14), Projection Pass is executed to fill the Intermediate Buffer. Next, our application passes the information to Reflection Pass in a fragment shader. After calculation of the screen space reflections is completed, the original scene image and the result are combined and output to the monitor.

4. Pixel-projected Screen Space Reflections

This chapter describes the main algorithm of this thesis. Key GLSL code snippets are provided on **Appendix A**.

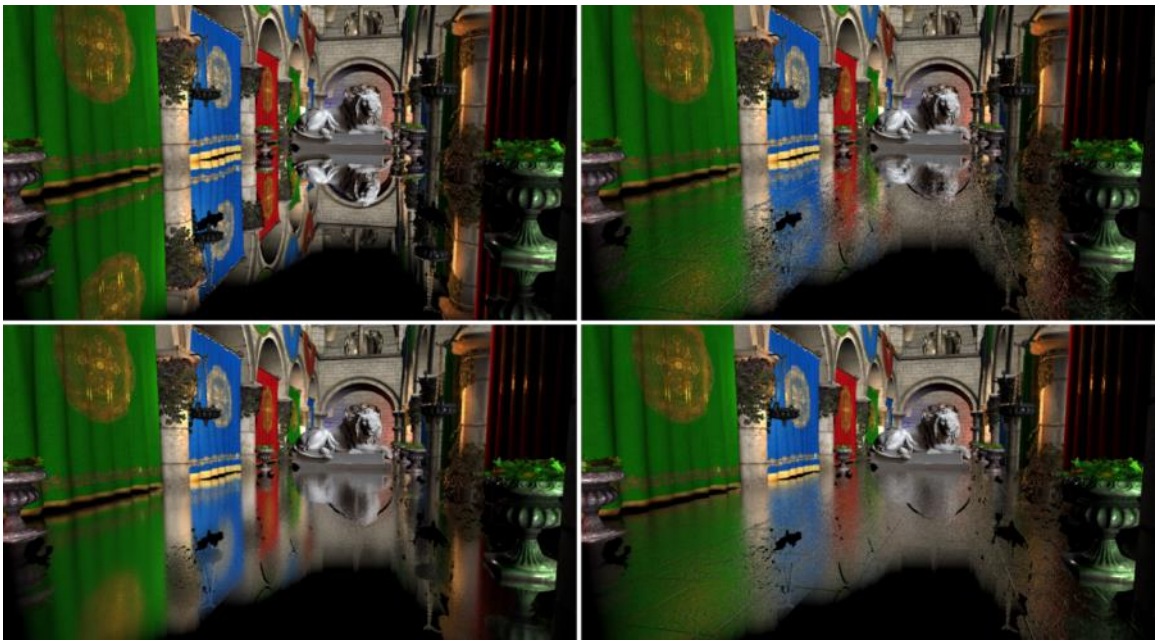

Figure 15: Image quality and performance comparison between individual statuses. Top Left: mirror planar reflection. Bottom Left: planar reflection with GGX Importance sampling. (4 Samples with Roughness 0.5) Top Right: planar reflection applied normal map. Bottom Right: planar reflection applied normal map with GGX Importance sampling. (4 Samples with Roughness 0.5)

4.1. Algorithm overview

This approach has two steps. First, in the “Projection pass”, for every pixel in screen space, find the reflectors which reflect the pixel. To do this, for each reflector, mirror the pixel in the world space across the reflector and do ray-tracing from main camera. If the pixel intersects in the bounds of a reflector, write the pixel’s color (reflected color) data in that place to the intermediate buffer. This does create an ordering issue which will be resolved later. Next, the “Reflection pass”, simply decodes pixel-data and obtains the reflected color from intermediate buffer.

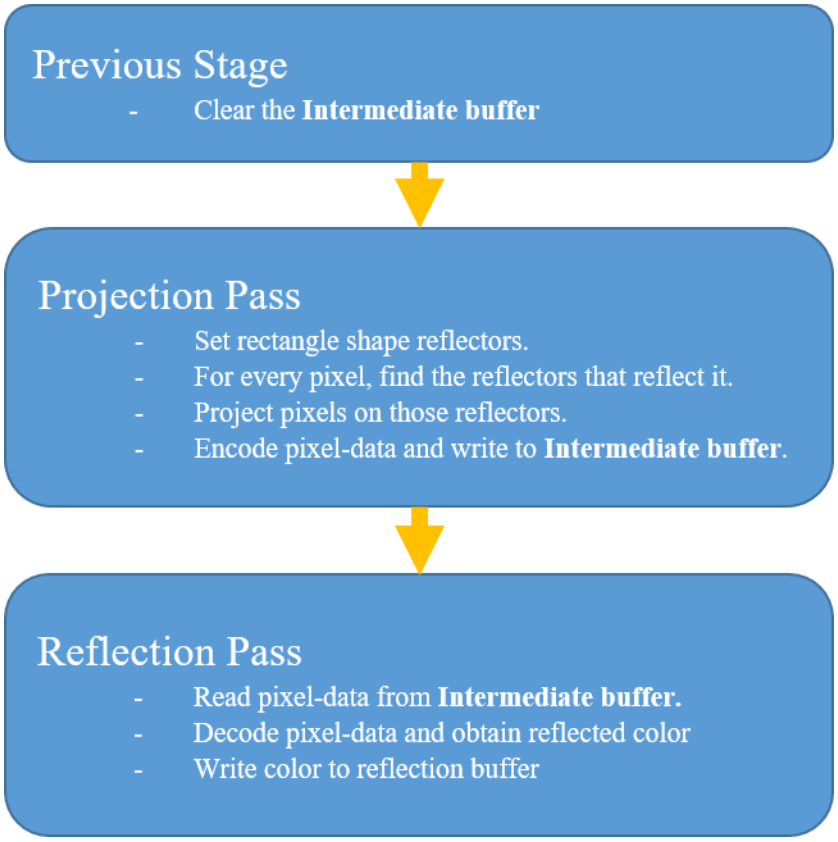

Figure 16: Algorithm of Pixel-projected reflections. (This figure based on [Cichocki17])

4.2. Previous Stage

To transfer information between the projection and reflection passes, we need to use a read/writeable buffer, which is called “Intermediate buffer”, which has the same number of elements as the number of pixels in the current screen. After creating the Intermediate buffer, clear the buffer before entering the Projection pass.

4.3. Projection Pass

4.3.1. Ray tracing

For simplicity, let's first look at the case where there is only one object and one reflector. First, every pixel in screen space calculates where it is reflected on the screen. (**Fig. 17**) To do this, mirror pixel position against normal of reflector and ray-cast toward mirrored pixel position (In this case, the reflector is a plane). If pixel's reflection is visible in the reflector, write the reflected color data in that place (Intermediate buffer). Otherwise, reject the pixel. But, what if the reflector was occluded by some other reflectors? Then, test other reflectors' bounds during ray-cast and reject the pixel if it is occluded.

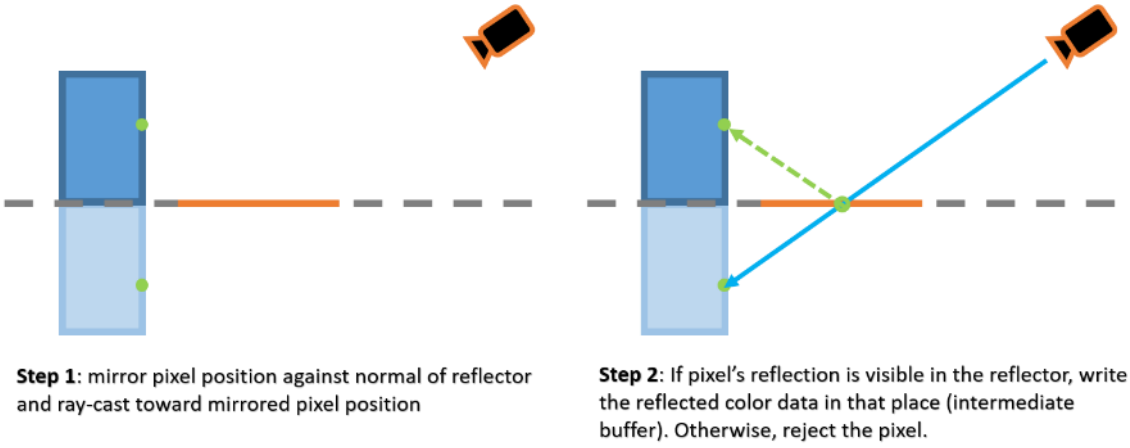

Figure 17: Illustration of the two steps for checking the visibility of a pixel on a reflector. (This figure based on [Cichocki17])

If we can just write the reflected color to Intermediate buffer in Projection pass and fetch it in Reflection pass, it would be the best situation. However, we have not considered an important exception. That is, multiple pixels can be written to the same place. (**Fig. 18**) That means we need to write the closest pixel to be reflected, only. But, we are

processing pixels, independently. And writing order between the threads of a shader depends on GPU scheduling.

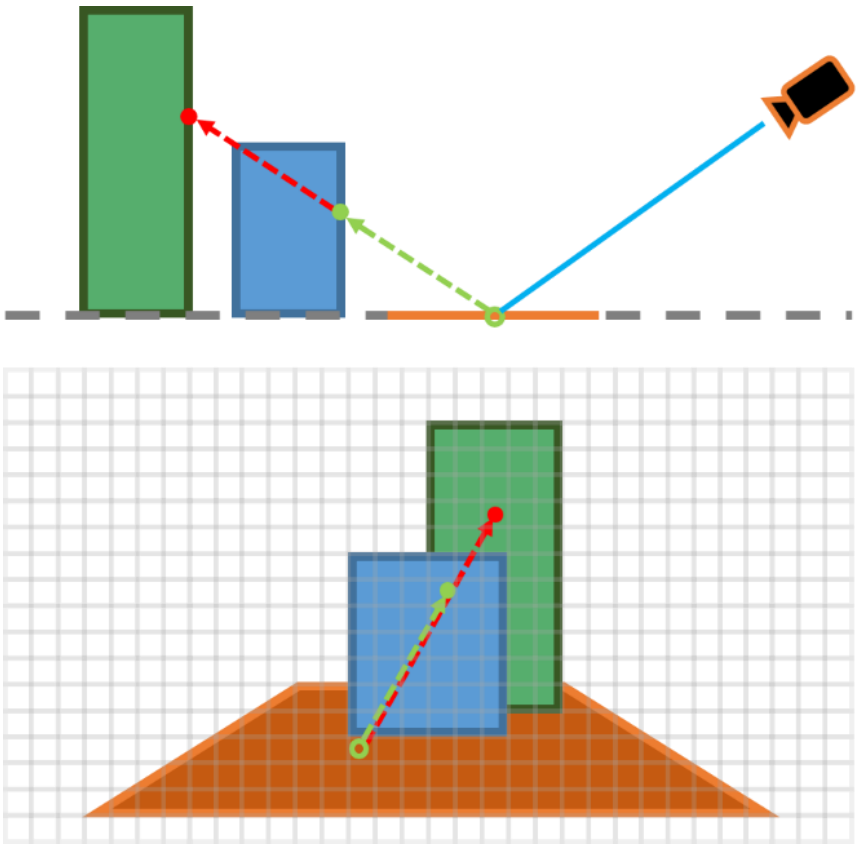

Figure 18: If multiple pixels are written to the same place, we need to write the closest pixel to be reflected. Top: view in side. Bottom: view in screen space.

4.3.2. Encoding Intermediate Buffer

To address the issues above, [Cichocki17] packed each offset data on the screen space between the reflected pixel and the reflecting pixel to one 32bit data. Because, we can solve the order issue with using atomic function (atomicMin function in this case). Then, how do we pack the offset data? To write the closet reflected pixel, we need to set certain offset coordinates system (2D) to compare which offset is the closest among the pixels written to the same place.

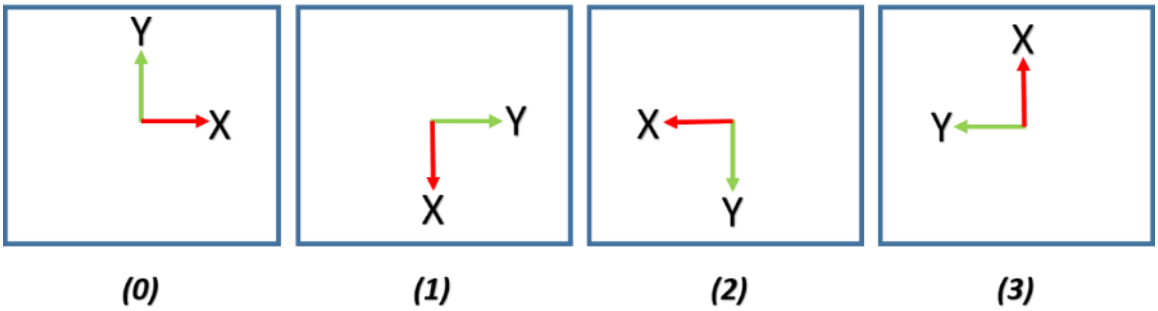

Figure 19: Four coordinate systems for Intermediate buffer encoding. (This figure based on [Cichocki17])

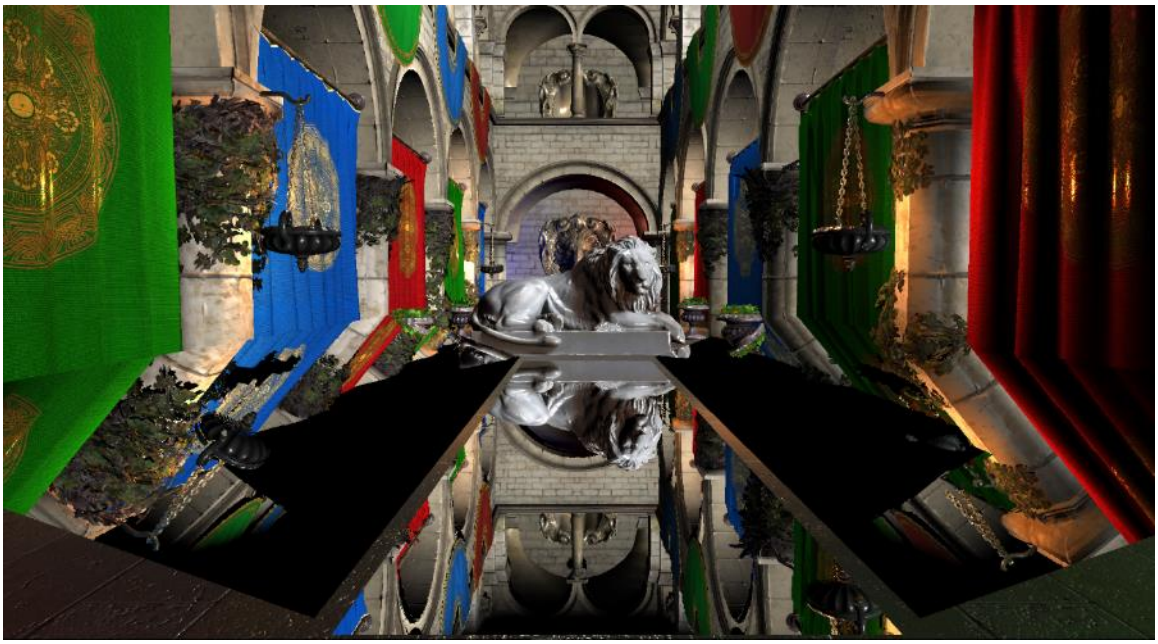

Figure 20: Three different orientations encoded in Intermediate buffer.

[Cichocki17] used 4 offset coordinate systems as shown (**Fig. 19**). First, find the longest axis among x and y-axis with its orientation positive or negative. Then, make the longest axis become y-axis. Lastly, depending on the new y-axis' orientation, x-axis' direction is determined. For example, if the longest axis of offset data is x-axis and its orientation is negative, the coordinate system is (3) in **Fig 19**. Now, from these new coordinate systems, 'y' coordinate is always positive. Thus, if we encode 'y' coordinate to most significant bits in one element of the Intermediate buffer, it can write the shortest (the

closet) pixel to the buffer with using **atomicMin** function. Intermediate buffer layout is as shown (**Fig. 21**). ([Cichocki17] uses the offset fractions for filtering, but this thesis will not cover it)

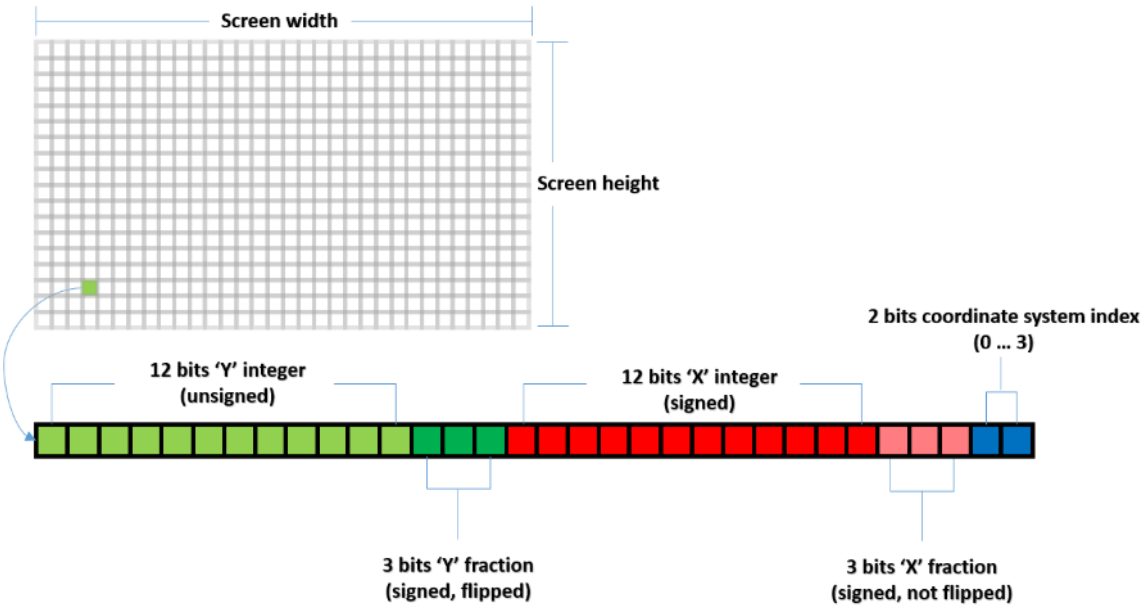

Figure 21: Intermediate buffer layout.

Unlike HLSL in DirectX, GLSL in OpenGL or Vulkan does not support float type atomic functions (only signed and unsigned integer data type are supported). Thus, we need to use **uint atomicMin** function for storing 32-bit unsigned int. And, we can use the Intermediate buffer type as a shader storage buffer object (SSBO) or a storage image. But, with SSBO, the cost will increases significantly as the buffer size increases. This is because the array retrieving time increases as the index of the array element increases. (SSBO which has 2,073,600 array length is used for FHD resolution 1,920 x 1,080) This is the same principle that the method using texture for storing tremendous number (over one million) of particles' information faster than the method using vertex

transformation in implementation of the GPU particle system. Therefore, we recommend to use a 32-bit unsigned storage image buffer with **uint imageAtomicMin** function.

4.4. Reflection Pass

This step is simple. For each pixel, read the Intermediate buffer and decode the data to retrieve the offsets. Then, obtain reflected color from the scene image. However, the result will have a lot of holes.

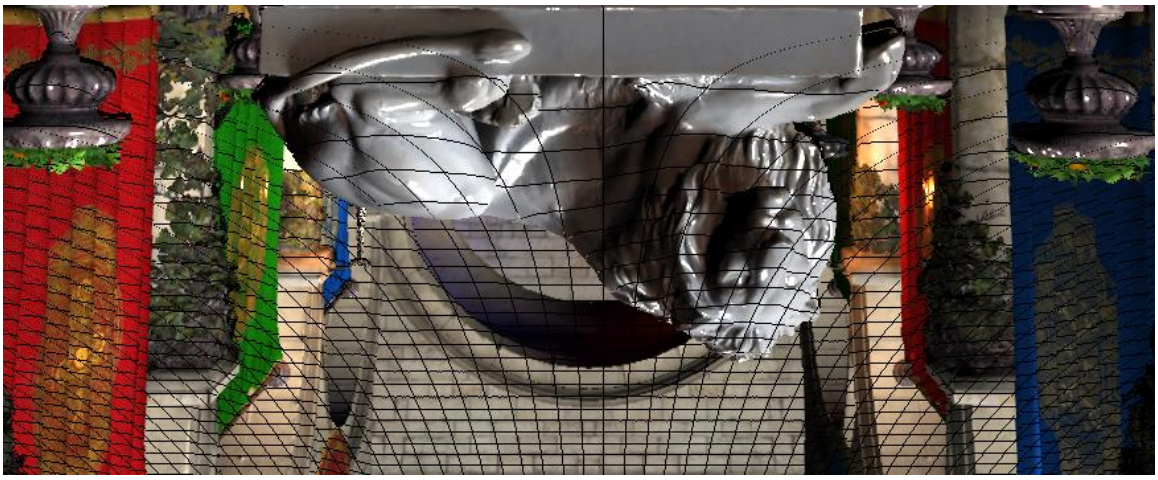

Figure 22: The illustration of the result without holes patching.

Because, the original scene image can get stretched when reflected. Thus, Intermediate buffer can get two types of missing data called holes in here. There are two types of holes. First one is the hole without any data. Second one is the hole which could not get data from first reflection layer. (**Fig. 23**) Fortunately, these holes do not form big groups. Thus, we can fill the holes like comparing neighboring pixels' offset and use the one which has the minimum offset among them.

Figure 23: Left: Two types of holes. In red circle, the holes have no data. In yellow circle, the holes which could not get data from first reflection layer. Right: After patching holes.

4.5. Apply Normal map

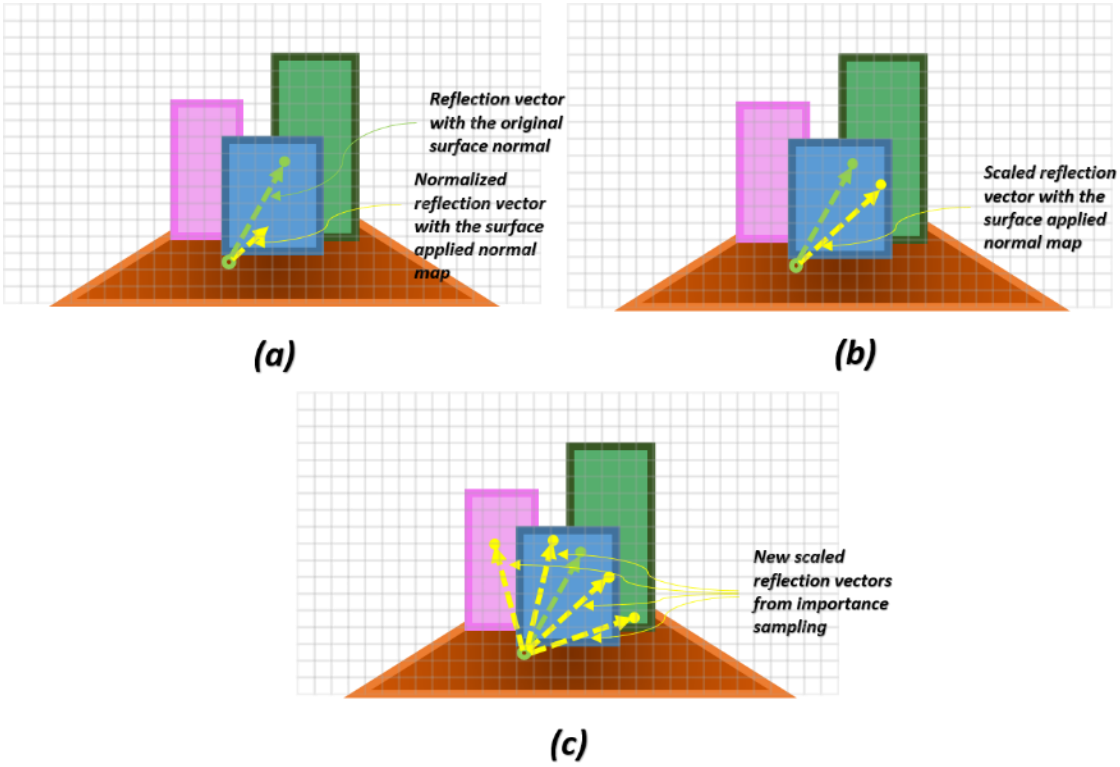

Figure 24: The steps for applying normal map and roughness on a reflector.

This approach uses ray tracing based on the fact that the normal of the reflector does not change. Therefore, it is impossible to apply normal maps to generate new reflection vector. The idea from original approach proposes an approximated approach to apply the normal map. In the reflection pass, find the distance between the reflected pixel and the reflecting pixel in the reflector in world space. Then, obtain a new reflection vector from a new normal vector calculated with the normal map. (**Fig. 24 - (a)**) Then, scale the new reflection vector by the original reflection distance we obtained earlier and add it at the reflecting position in world space. (**Fig. 24 - (b)**) Lastly, project the position to screen space and fetch the color from the original scene image.

In most situations, the result looks good. But, if the view vector is getting closer in parallel with the reflector, the distance error between the actual reflected pixel and the pixel indicated by the new reflection vector is intensified. Therefore, the distortion of the result output becomes worse.

4.6. Approximate Reflections along the Roughness

So far, we have only dealt with mirror reflections and could not present scattered reflections along surface (reflector) roughness. To improve this, inspired by [Stachowiak15], we used a Monte Carlo method to approximate the result using samples with multiple random seed values. First, generate random seed Xi, which is consists of two values of which ranges are [0, 1], per sample. Then, do importance sampling (with the seed Xi) according to the lighting model we are currently using to obtain the reflection vector. If the reflection vector is valid, as what we did to apply Normal map, scale the reflection vector and fetch the values from where each reflection vectors indicate and average them. (**Fig. 24 - (c)**) (In this thesis, it uses 4 samples with GGX importance sampling)

5. Results

I implemented this approach on an NVIDIA GTX 1060 (Core Speed: 1404 MHz, Memory Speed: 2002 MHz, Memory Bus Width: 192 Bit) with an Intel Core i7-6700HQ Quad Core Processor (2.6 GHz). I tested the approach on a complex scene which uses Crytek's Sponza and a Lion modeling designed by Geoffrey Marchal. Every case is tested in full HD resolution 1920 x 1080. For convenience, we use the term SSR as screen space reflections.

5.1. Performance of each stage

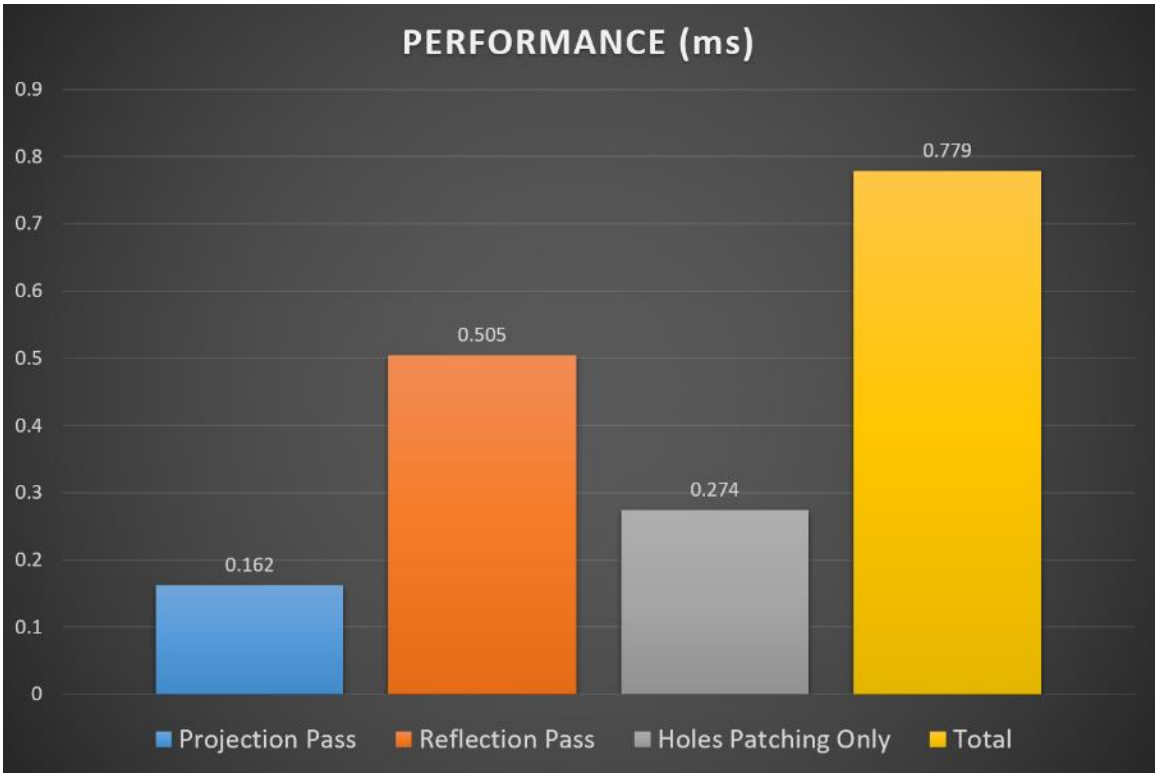

Chart 1: Performance test for each stage. This result is from standard settings (using one reflector which covers whole floor of the scene, without normal map and importance sampling).

5.2. Comparison with different number of reflectors

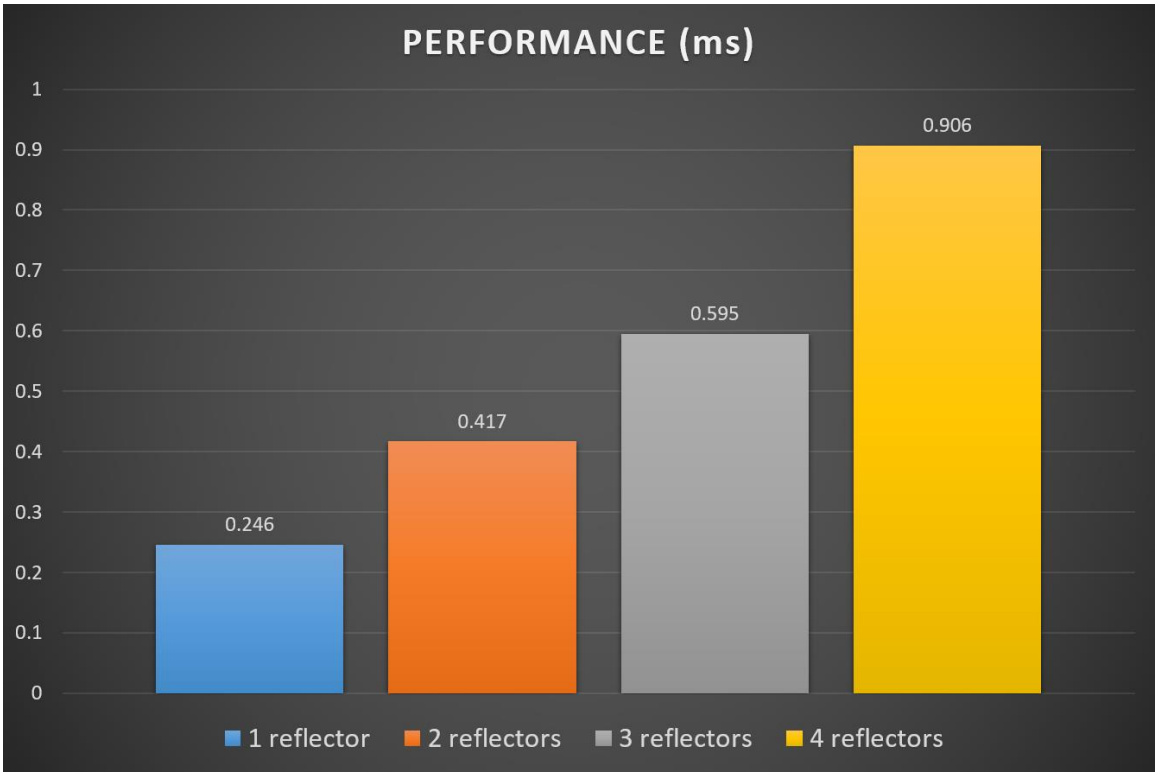

Chart 2: Performance test with different number of reflectors. It is more important the fact how large the reflectors fill the screen than the additional cost for increased number of the reflectors.

5.3. Comparison with Brute Force Screen Space Reflections

5.3.1 Mirror Reflections

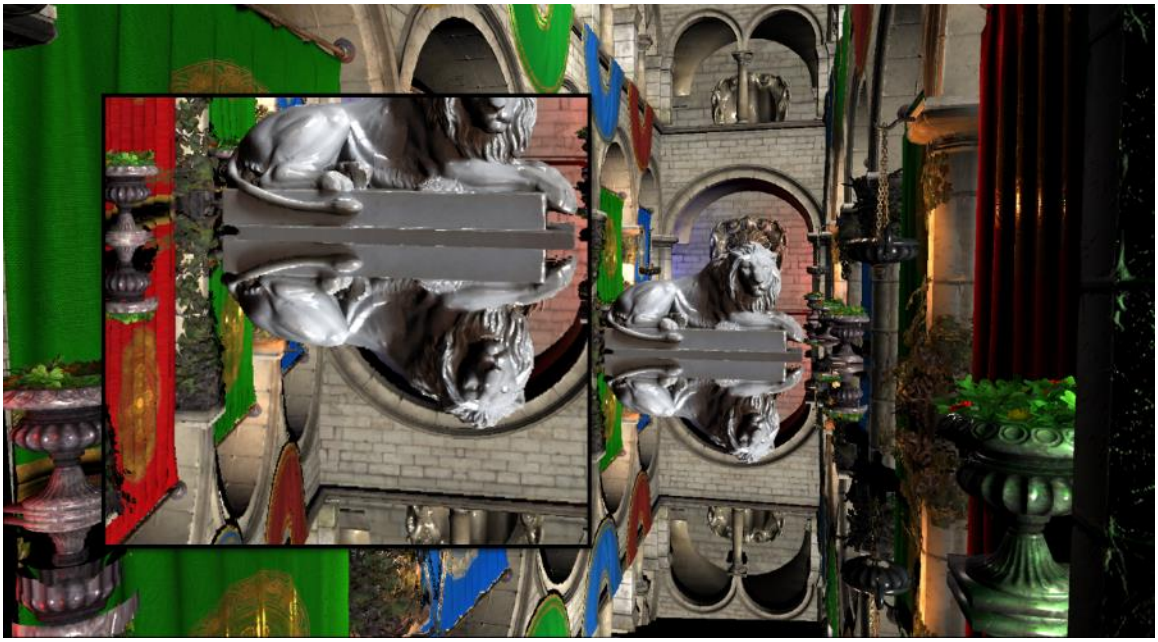

Figure 25: Brute force SSR (High quality). Max iteration: 1024. Step size: 0.1. (17.387ms)

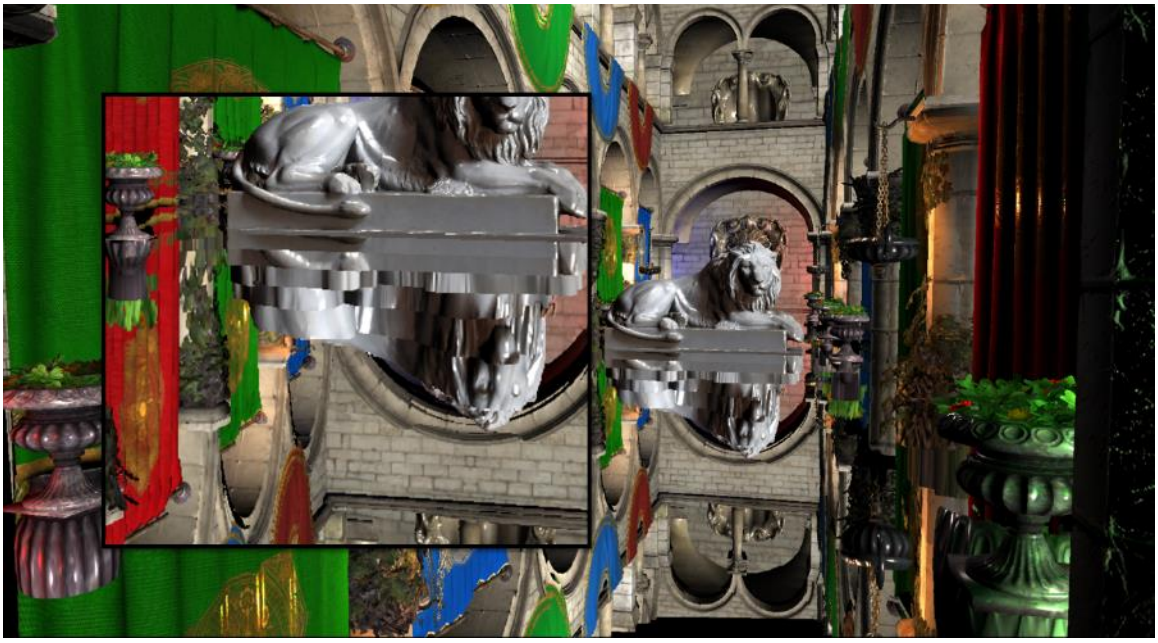

Figure 26: Brute force SSR (Low quality). Max iteration: 128. Step size: 5.0. (1.071ms)

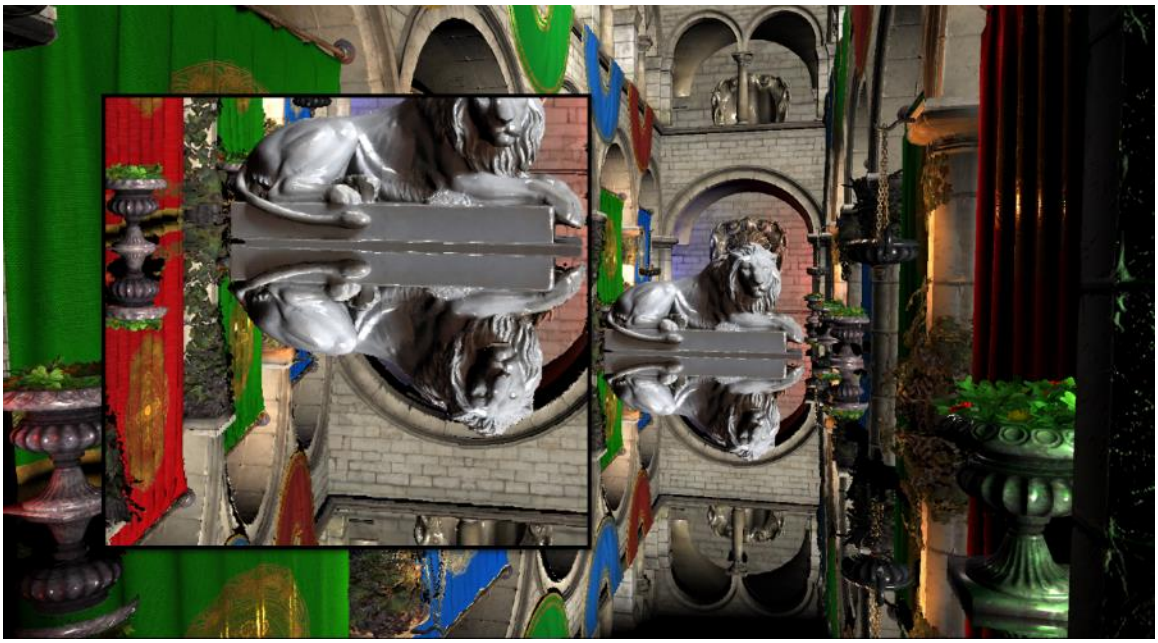

Figure 27: Pixel-projected SSR. (0.779ms)

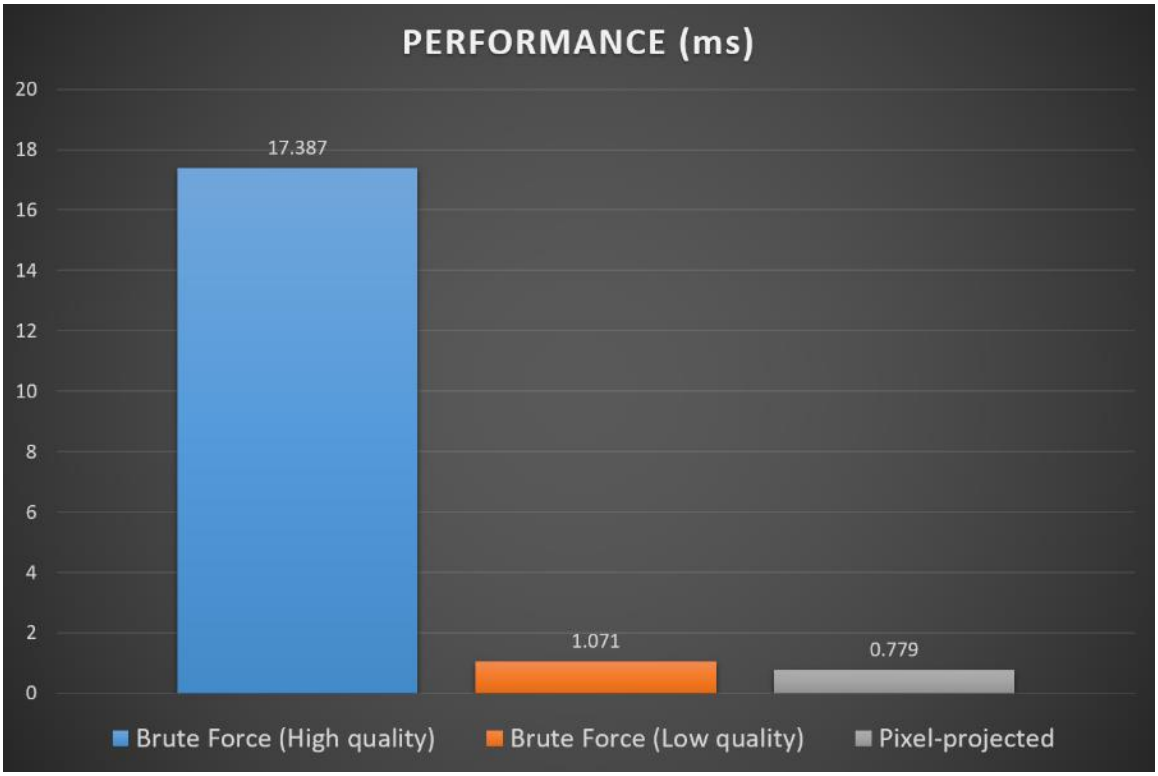

Chart 3: Performance comparison between Brute force SSR and Pixel-projected SSR.

5.3.2 Reflections with Normal map

Figure 28: Brute force SSR (High quality). Max iteration: 1024. Step size: 0.1. Using depth linear interpolation. Using normal map. (31.071ms)

Figure 29: Brute force SSR (Low quality). Max iteration: 128. Step size: 5.0. Using depth linear interpolation. (1.941ms)

Figure 30: Pixel-projected SSR. Using normal map. (1.306ms)

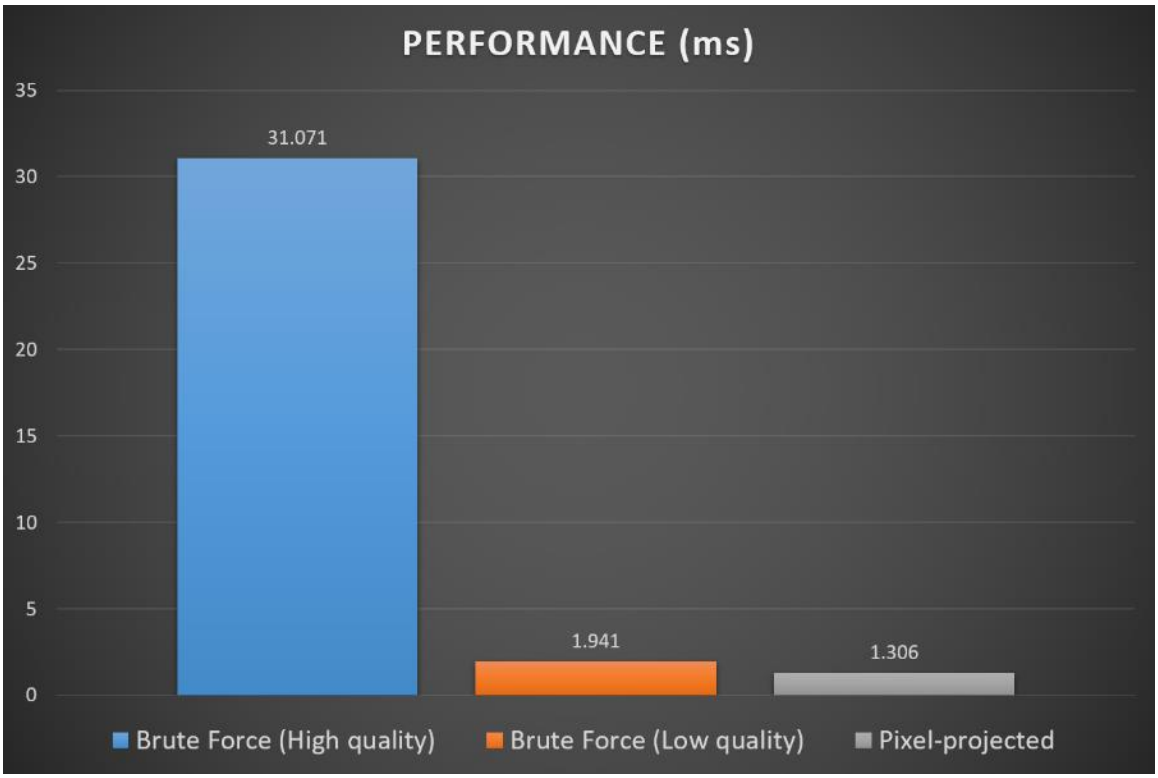

Chart 4: Performance comparison between Brute force SSR and Pixel-projected SSR with using normal map.

5.4. Comparison of the results with various Roughness

5.4.1 Mirror Reflections

Figure 31: Pixel-projected SSR. Roughness: 0.2. GGX Importance sampling. Using 4 samples. (0.802 ms)

Figure 32: Pixel-projected SSR. Roughness: 0.5. GGX Importance sampling. Using 4 samples. (1.151 ms)

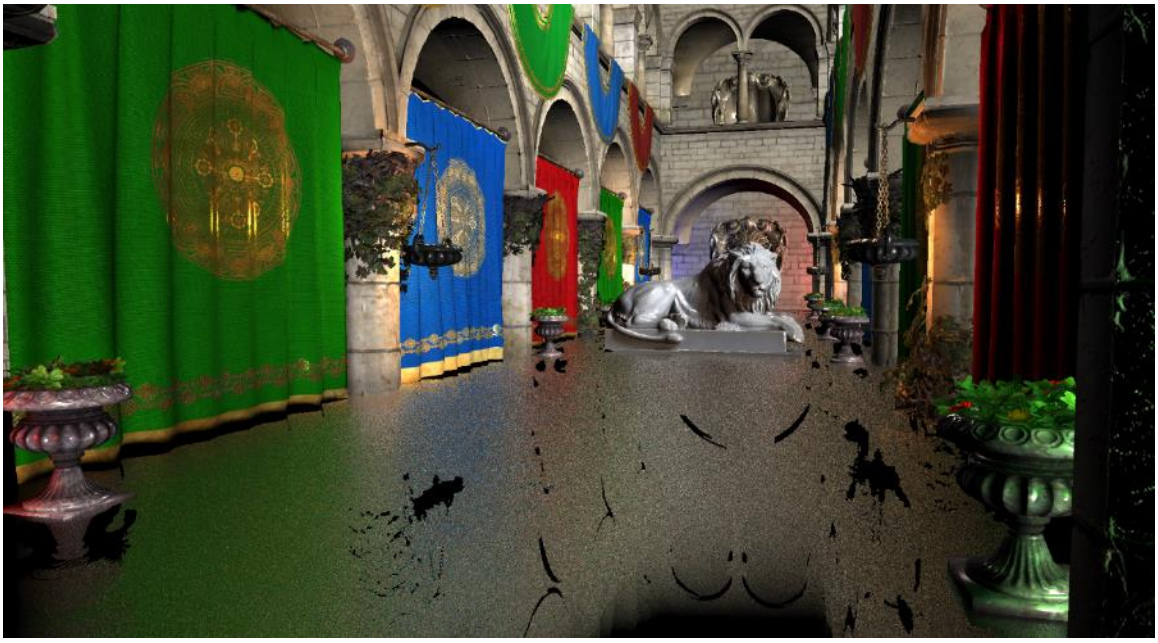

Figure 33: Pixel-projected SSR. Roughness: 1.0. GGX Importance sampling. Using 4 samples. (1.621 ms)

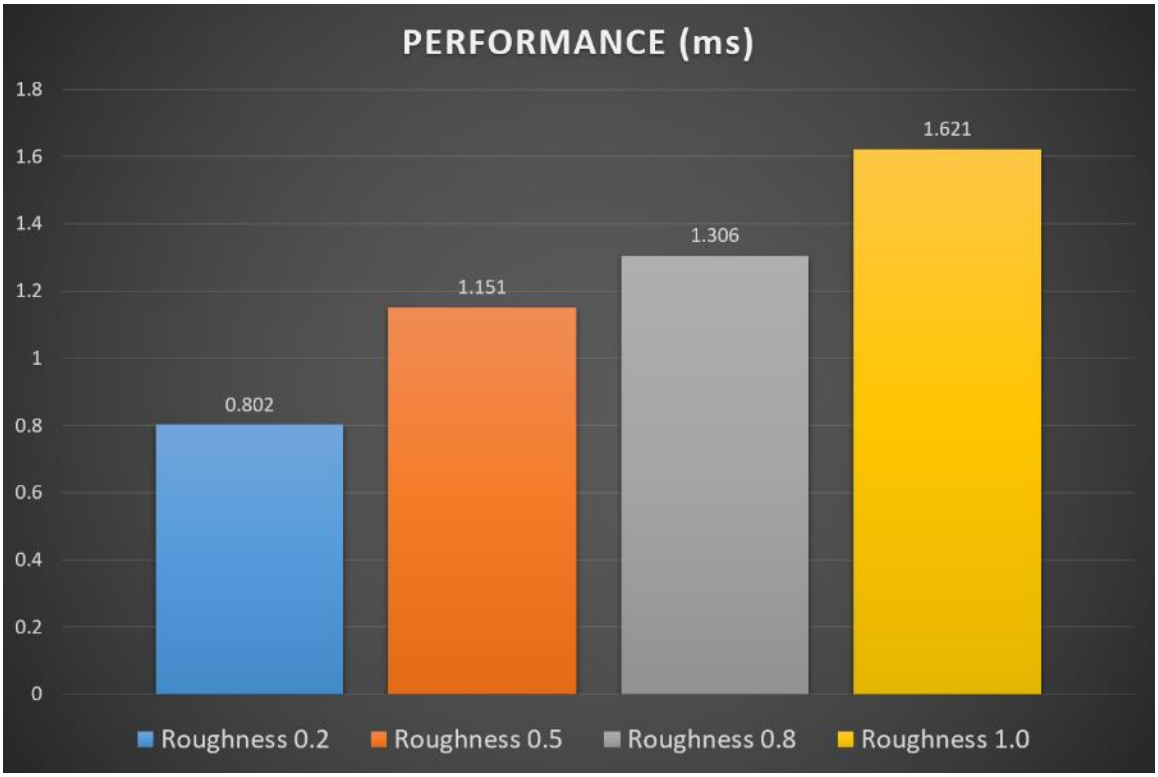

Chart 5: Performance comparison of Pixel-projected SSR with various roughness.

5.4.2 Reflections with Normal map

Figure 34: Pixel-projected SSR. Roughness: 0.0. GGX Importance sampling. Using 4 samples. Using normal map. (1.306 ms)

Figure 35: Pixel-projected SSR. Roughness: 0.2. GGX Importance sampling. Using 4 samples.Using normal map. (1.505 ms)

Figure 36: Pixel-projected SSR. Roughness: 0.5. GGX Importance sampling. Using 4 samples. Using normal map. (1.784 ms)

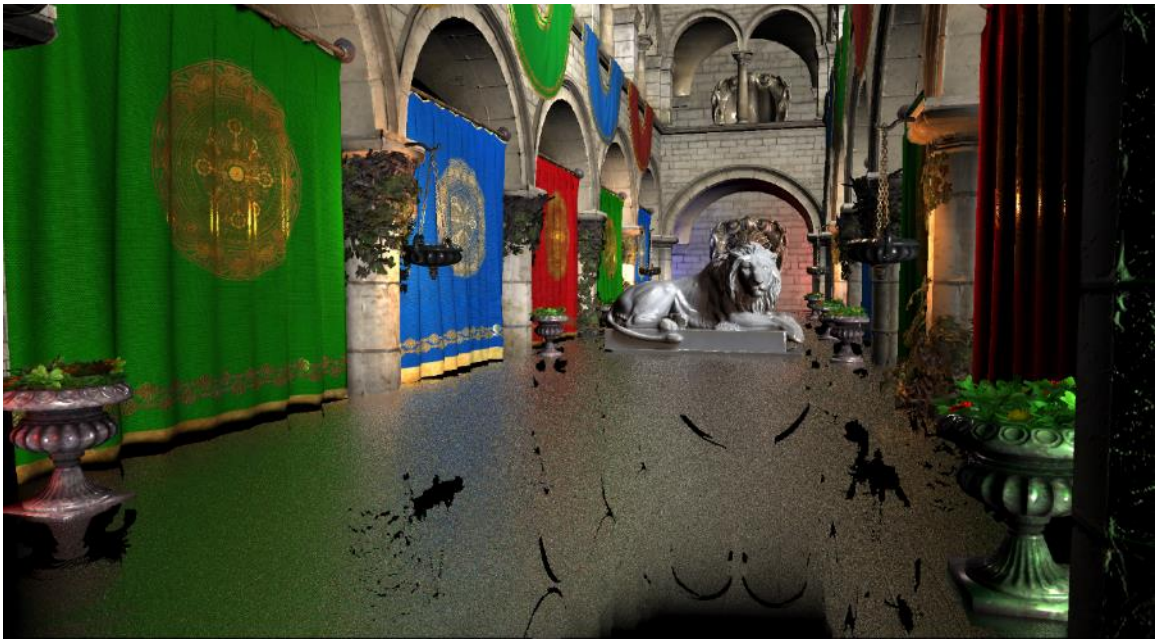

Figure 37: Pixel-projected SSR. Roughness: 1.0. GGX Importance sampling. Using 4 samples. Using normal map. (1.946 ms)

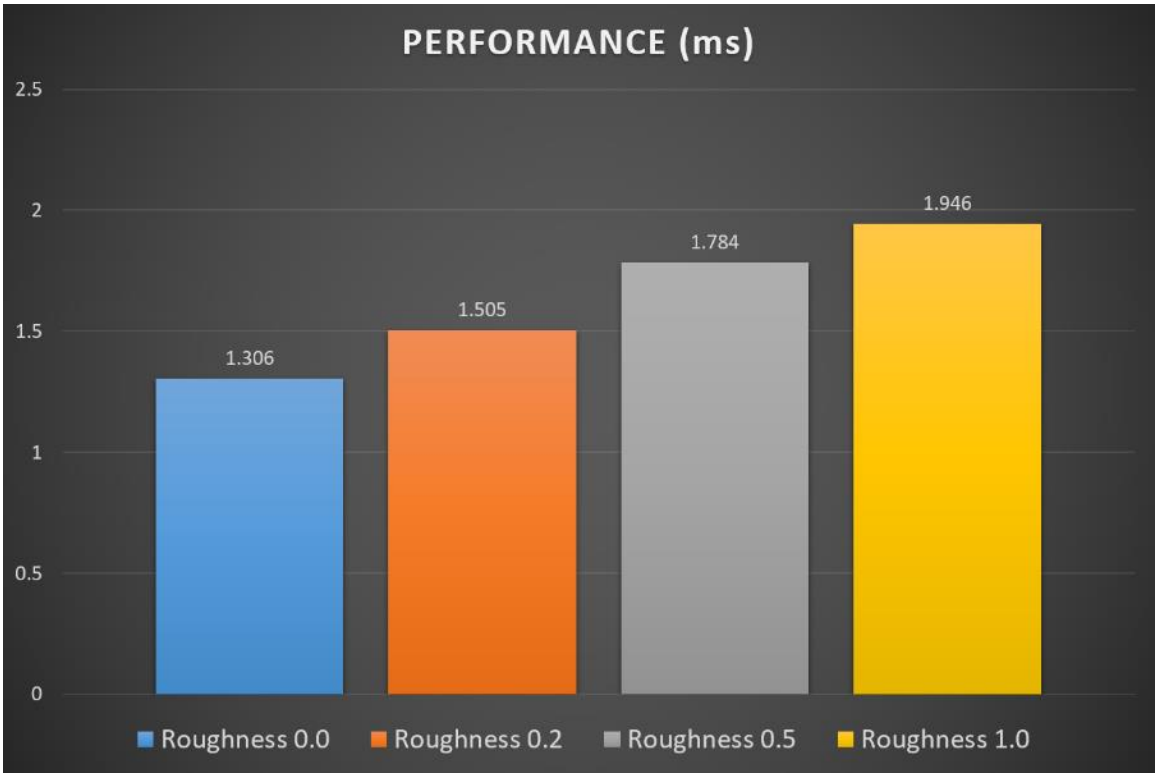

Chart 6: Performance comparison of Pixel-projected SSR using normal map with various roughness.

5.5. Comparison of the results with various number of Samples

Figure 38: Pixel-projected SSR. Roughness: 0.5. GGX Importance sampling. Using 1 samples. (1.498 ms)

Figure 39: Pixel-projected SSR. Roughness: 0.5. GGX Importance sampling. Using 4 samples. (1.678 ms)

Figure 40: Pixel-projected SSR. Roughness: 0.5. GGX Importance sampling. Using 8 samples. (1.764 ms)

Figure 41: Pixel-projected SSR. Roughness: 0.5. GGX Importance sampling. Using 16 samples. (2.231 ms)

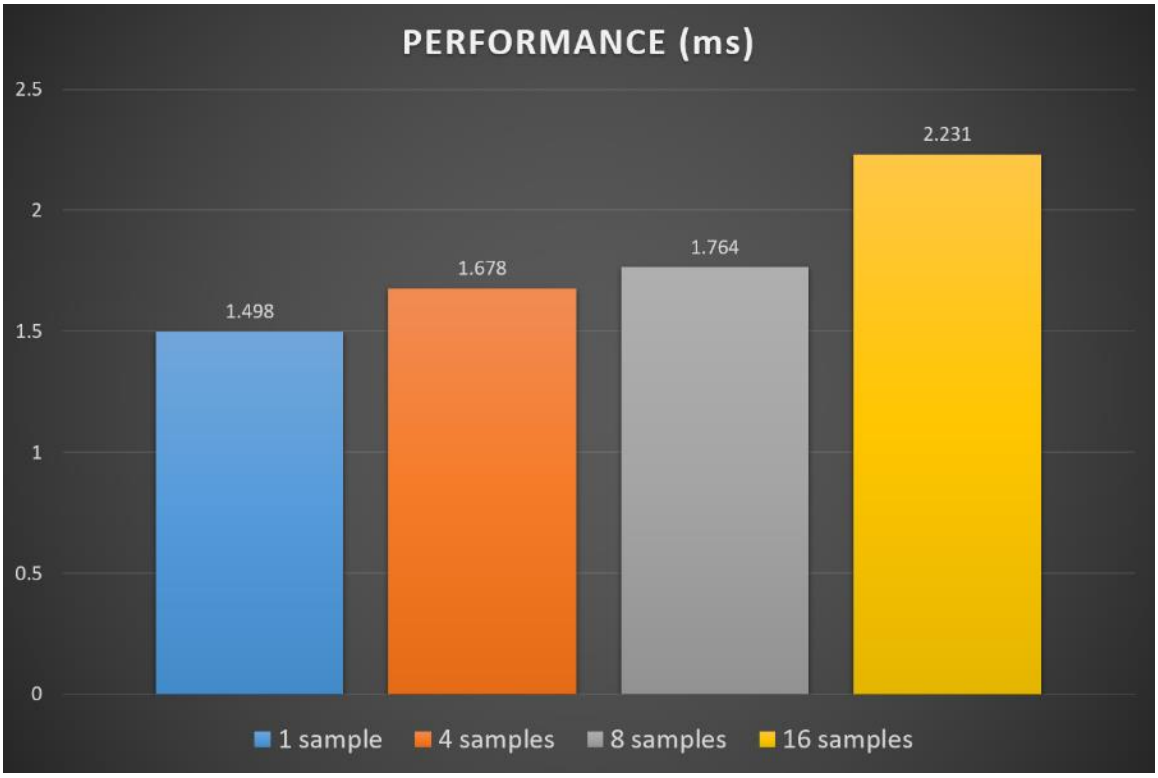

Chart 7: Performance comparison of Pixel-projected SSR with various number of samples.

6. Limitations

One of the most representative limitations of screen space reflections is that geometries not drawn on the original scene image cannot be reflected. Pixel-projected reflections have same limitation. There are two cases which lead this phenomenon. First case is that the geometry itself is in the screen space, but it is not drawn on the original scene image because it is occluded by other geometries. Second case is when the geometry is not in the bound of the original scene image. In the latter case, the fade effect is applied to minimize the artifact.

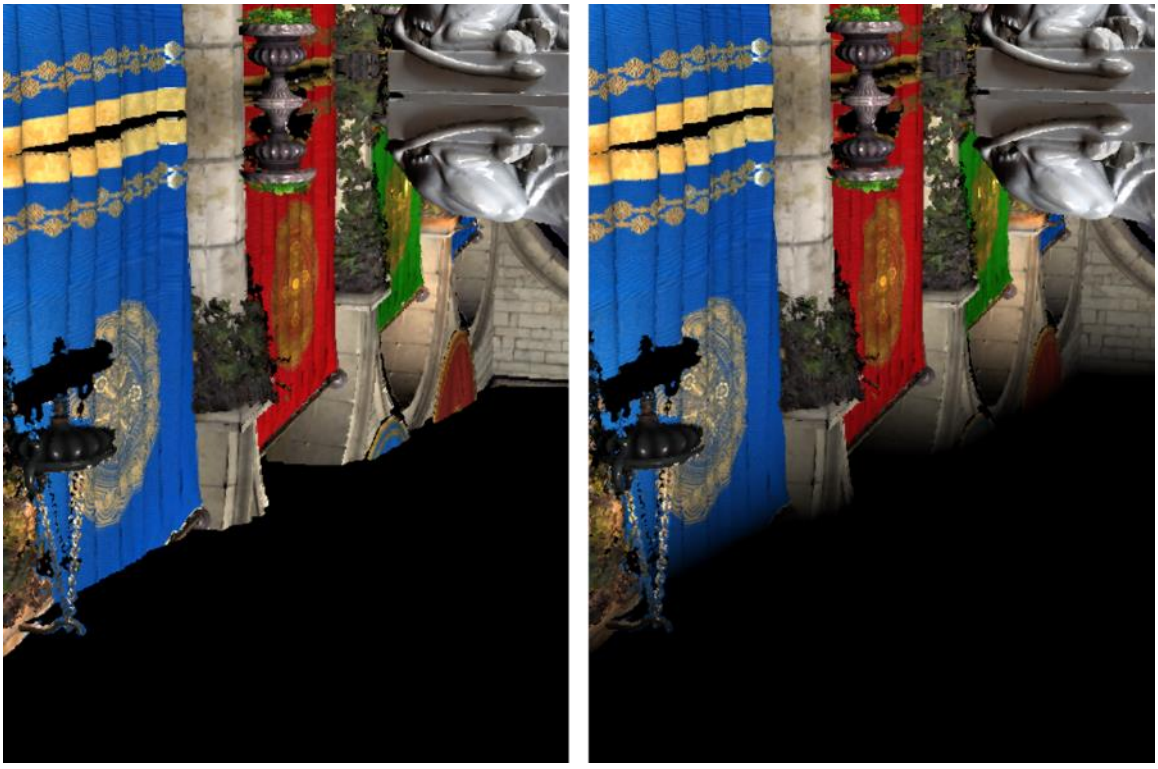

Figure 42: The difference results with or without fade effect. Left: Without fade effect. Right: With fade effect.

Also, to account for the roughness of reflectors, this approach relies on approximations to get final colors from the original scene image. As the roughness increases, the probability of fetching the color information of the irrelevant places also increases, which causes color bleeding artifacts.

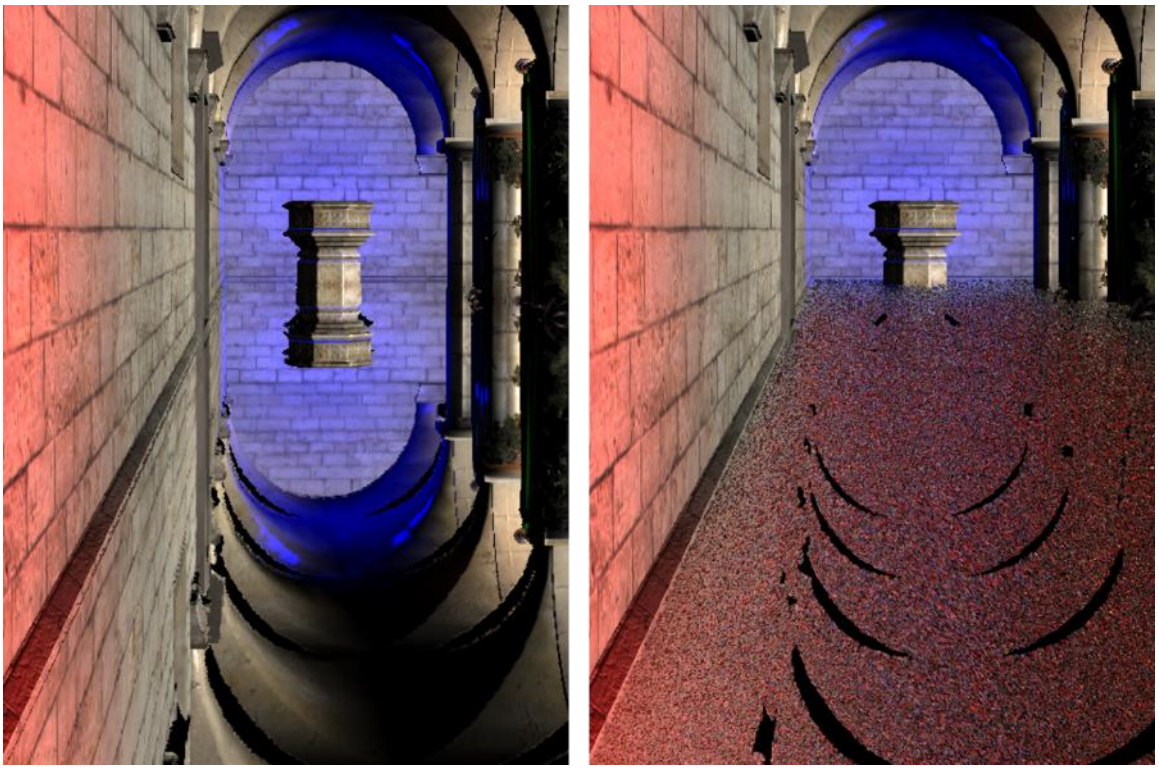

Figure 43: As the roughness of reflectors increases, the probability of color bleeding artifacts also increases. Left: Roughness 0.0. Right: Roughness 1.0. GGX Importance sampling. Using 4 samples.

And, due to use the original image which already contains the results of the lighting calculation, it cannot reflect glossy reflections.

Lastly, this approach should only be used for simple geometric primitives. Using reflectors for complex geometries increases the computation for ray tracing, which can seriously degrade the performance.

7. Conclusion and Future Work

In terms of visual quality, Pixel-projected Screen Space Reflections is much better than low quality Brute force Screen Space Reflections which have similar performance. The result is also better than those of the high quality Brute force Screen Space Reflections we tested. Thus, in the case of mirror reflections, Pixel-projected Screen Space Reflections is a superior method in terms of quality and performance.

When using a normal map, mirror reflection shows similar results for Brute force Screen Space Reflections and Pixel-projected Screen Space Reflections. However, Brute force Screen Space Reflections' missing data area is somewhat blurred by the variation of the direction of the rays from normal map. Since the missing data area of Pixel-projected Screen Space Reflection does not change, the black holes created by the projection are not affected by the normal map.

As the roughness of the reflector increases, the color of reflections also becomes blurred, but the missing data area also remains unchanged. The performance also decreases as the roughness increases because the probability of fetching pixels farther away from the current pixel increases, which is less memory-coherent.

As expected, the greater the number of importance samples, the smoother the results, but the higher the costs. In terms of quality, it is hard to notice any improvement after 8 samples.

In conclusion, Pixel-projected Screen Space Reflections achieves more accurate reflection images than previous approaches with lower cost. It only guarantees high performance with a simple reflector shapes in real-time environments, but provides high-

quality results in common scenes. In the future, if more efficient ray tracing solutions for complex geometry on GPU are devised, this approach could become a much more popular algorithm for screen space reflections.

Incidentally, Microsoft presented DirectX Raytracing, a next generation graphics API for ray tracing, in GDC 2018. With this, it is expected that the use of an efficient GPU ray tracing acceleration structure from DirectX Raytracing will enable ray traversal of complex geometry with much lower cost than before. In other words, Pixel-projected reflections using complex reflectors beyond simple planar reflectors are expected to be possible in real-time environments in the near future.

8. Appendix A: GLSL codes for key parts

8.1. Encoding offset data for Intermediate Buffer in Projection Pass

```
uint packInfo(vec2 offset)
{
	uint CoordSys = 0;

	uint YInt = 0;
	int YFrac = 0;
	int XInt = 0;
	int XFrac = 0;

	//define CoordSystem
	if (abs(offset.y) < abs(offset.x))
	{
		if (offset.x < 0.0)
		{
			YInt = uint(-offset.x);
			YFrac = int(fract(offset.x)*8.0);
			XInt = int(offset.y);
			XFrac = int(fract(offset.y)*8.0);

			CoordSys = 3;
		}
		else
		{
			YInt = uint(offset.x);
			YFrac = int(fract(offset.x)*8.0);
			XInt = int(offset.y);
			XFrac = int(fract(offset.y)*8.0);

			CoordSys = 1;
		}
	}
	else
	{
		if (offset.y < 0.0)
		{
			YInt = uint(-offset.y);
			YFrac = int(fract(offset.y)*8.0);
			XInt = int(offset.x);
			XFrac = int(fract(offset.x)*8.0);

			CoordSys = 2;
		}
		else
		{
			YInt = uint(offset.y);
			YFrac = int(fract(offset.y)*8.0);
			XInt = int(offset.x);
			XFrac = int(fract(offset.x)*8.0);

			CoordSys = 0;
		}
	}

	return  ((YInt & 0x00000fff) << 20) | ((YFrac & 0x00000007) << 17) | ((XInt & 0x00000fff) << 5) |
		((XFrac & 0x00000007) << 2) | CoordSys;
}
```

8.2. Decoding uint data in Reflection Pass

```
vec4 unPacked(in uint unpacedInfo, in vec2 dividedViewSize, out uint CoordSys)
{
	float YInt = float(unpacedInfo >> 20);
	int YFrac = int((unpacedInfo & 0x000E0000) >> 17);

	uint uXInt = (unpacedInfo & 0x00010000) >> 16;

	float XInt = 0.0;

	if (uXInt == 0)
	{
		XInt = float(int((unpacedInfo & 0x0001FFE0) >> 5));
	}
	else
	{
		XInt = float(int(((unpacedInfo & 0x0001FFE0) >> 5) | 0xFFFFF000));
	}

	int XFrac = int((unpacedInfo & 0x0000001C) >> 2);

	float Yfrac = YFrac * 0.125;
	float Xfrac = XFrac * 0.125;

	CoordSys = unpacedInfo & 0x00000003;

	vec2 offset = vec2(0.0);

	if (CoordSys == 0)
	{
		offset = vec2((XInt) / dividedViewSize.x, (YInt) / dividedViewSize.y);
	}
	else if (CoordSys == 1)
	{
		offset = vec2((YInt) / dividedViewSize.x, (XInt) / dividedViewSize.y);
	}
	else if (CoordSys == 2)
	{
		offset = vec2((XInt) / dividedViewSize.x, -(YInt) / dividedViewSize.y);
	}
	else if (CoordSys == 3)
	{
		offset = vec2(-(YInt) / dividedViewSize.x, (XInt) / dividedViewSize.y);
	}

	return vec4(offset, Xfrac, Yfrac);
}
```

8.3. GGX Importance Sampling

```
vec3 GGXDistribution_Sample_wh(vec2 xi, float roughness)
{
	vec3 wh;
	float phi = (2.0 * PI) * xi.y;

	float tanTheta2;
	float cosTheta;

	tanTheta2 = roughness * roughness * xi.x / (1.0 - xi.x);
	cosTheta = 1.0 / sqrt(1.0 + tanTheta2);

	float sinTheta = sqrt(max(0.0, 1.0 - cosTheta * cosTheta));

	wh = vec3(sinTheta * cos(phi), sinTheta * sin(phi), cosTheta);
	return wh;
}


//wo -> viewvector in tangentSpace
vec3 ImportanceSampleGGX(vec3 normal, vec3 worldView, vec2 xi, float roughness, mat3 rotationMat)
{
	vec3 wh = normalize(GGXDistribution_Sample_wh(xi, roughness)); //tangent_Space

	vec3 tangentNormal = vec3(0.0, 0.0, 1.0);

	if (tangentNormal == normal || tangentNormal == -normal)
	{
		//tangent space to world space
		vec3 worldHalf = rotationMat * wh;
		worldHalf = normalize(worldHalf);

		//world reflection Vector
		return normalize(2.0 * dot(worldView, worldHalf)*worldHalf - worldView);
	}
	else
	{
		float dotValue = dot(vec3(0.0, 0.0, 1.0), normal);

		float angle = acos(dotValue);

		vec3 bitangentVec = normalize(cross(tangentNormal, normal));

		wh = mat3(rotationMatrix(bitangentVec, angle)) * wh;
		vec3 worldHalf = rotationMat * wh; //tan - > world
		worldHalf = normalize(worldHalf);

		//world reflection Vector
		return normalize(2.0 * dot(worldView, worldHalf)*worldHalf - worldView);
	}
}
```

8.4. Holes Patching

```
if (bValidPixel)
{
	//if the offset is greater than threshold, do not use the pixel for holes patching
	float threshold = thisColor.w;
	float minOffset = threshold;

	//find the closest neighbor pixel and fetch the color
	vec4 neighborColor00 = texture(SceneTexture, fragUV + vec2(1.0 / viewPortSize.x, 0.0));
	if (neighborColor00.w > 0.0)
	{
		minOffset = min(minOffset, neighborColor00.w);
	}

	vec4 neighborColor01 = texture(SceneTexture, fragUV - vec2(1.0 / viewPortSize.x, 0.0));
	if (neighborColor01.w > 0.0)
	{
		minOffset = min(minOffset, neighborColor01.w);
	}

	vec4 neighborColor02 = texture(SceneTexture, fragUV + vec2(0.0, 1.0 / viewPortSize.y));
	if (neighborColor02.w > 0.0)
	{
		minOffset = min(minOffset, neighborColor02.w);
	}

	vec4 neighborColor03 = texture(SceneTexture, fragUV - vec2(0.0, 1.0 / viewPortSize.y));
	if (neighborColor03.w > 0.0)
	{
		minOffset = min(minOffset, neighborColor03.w);
	}

	if (minOffset == neighborColor00.w)
	{
		outColor = neighborColor00;
	}
	else if (minOffset == neighborColor01.w)
	{
		outColor = neighborColor01;
	}
	else if (minOffset == neighborColor02.w)
	{
		outColor = neighborColor02;
	}
	else if (minOffset == neighborColor03.w)
	{
		outColor = neighborColor03;
	}

	//if the offset is equal or less or than threshold, do not use the pixel for holes patching
	if (minOffset <= threshold)
		outColor.w = 1.0;
	else
		outColor.w = 0.0;
}
```

9. Bibliography

[Akenine18] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michael Iwanicki: Real-Time Rendering, 4th Edition.

[Appel68] Arthur Appel: Some techniques for shading machine renderings of solids. AFIPS Conference Proc. 32 pp.37-45

[Bruce07] Bruce Walter, Steve Marschner, Hongsong Li, Ken Torrance: Microfacet Models for Refraction through Rough Surfaces.

[Cichocki17] Adam Cichocki: Optimized Pixel-Projected Reflections for Planar Reflectors, SIGGRAPH 2017.

[Giacaone16] Michele Giacalone: Screen Space Reflections in The Surge.

[Gregory14] Jason Gregory: Game Engine Architecture 2nd Edition.

[Lapinski17] Pawel Lapinski: Vulkan Cookbook.

[Le17] Trung Le: Evaluating BVH splitting strategies.

[Luna16] Frank D. Luna: 3D Game Programming with DirectX 12.

[Mansouri16] Jalal El Mansouri: Rendering Rainbow Six Siege, GDC, 2016.

[McGuire14] Morgan McGuire, Michael Mara: Efficient GPU Screen-Space Ray racing

[Pharr17] Matt Pharr, Wenzel Jakob, Greg Humphreys: Importance Sampling, Physically Based Rendering 3rd edition. pp.794-796.

[Raytracing18] “Wikipedia, Ray tracing (graphics)”. URL:
< https://en.wikipedia.org/wiki/Ray_tracing_(graphics)>

[Stachowiak15] Tomasz Stachowiak, Yasin Uludag: Stochastic Screen-Space, SIGGRAPH 2015.

[Uludag14] Yasin Uludag: Hi-Z Screen-Space Cone-Traced Reflections, GPU Pro 5.

[Valient13] Michal Valient: Killzone Shadow Fall Demo Postmortem.

[Valient14] Michal Valient: Reflections and Volumetrics of Killzone: Shadow Fall, SIGGRAPH, 2014.

[Vulkan17] “Vulkan Tutorial”.
< https://vulkan-tutorial.com>

[Wronski14] Bart Wronski: Assassin's Creed 4: Road to Next-gen Graphics, GDC, 2014.

10. Credits

The “Crytek Sponza” model is from McGuire Computer Graphics Archive

< http://casual-effects.com/data/>

The “Lion” model designed by Geoffrey Marchal is from Sketchfab

< https://sketchfab.com/models/455b37b9e58d43dba5b6c0ab269e242a>