CHALMERS

Approximating ray traced reflections using screen-space data

Master of Science Thesis in the Programme Computer Science: Algorithms, Languages and Logic

MATTIAS JOHNSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, April 2012

Approximating ray traced reflections using screen-space data

Mattias Johnsson

Examiner: Ulf Assarsson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden April 2012

1 Abstract

At Spark Vision, independent component images, each containing a part of the rendered geometry, are layered to create complete images. Due to the assumption of independence; reflections cannot be accurately rendered. A screen-space method for adding reflections to a rendered image using buffers for geometry and surface properties is proposed. Reflections are traced using an approximation of ray tracing. The goal is to allow for the continued use of component images with reflections added as a post-processing effect in real time.

The method outlined allows for perfect and glossy reflections comparable to the quality of commercial ray tracers for optimal scenes. However, it fails to capture reflections of objects that are not visible from the camera view-point except for cases in which these are part of a pre-defined surrounding environment. The method itself allows for dynamic scenes, camera settings and surface properties including BRDFs if used as an off-line renderer.

Due to demands of image quality, time constraints and the lack of GPU support a method for caching the rays is proposed based on assumption of a static geometry. The caching method allows for arbitrary variations in lighting, textures and reflection strength as well as limited variations of normal mapping and surface shininess. This method achieves acceptable running times for the intended application.

2 Sammanfattning

På Spark Vision används oberoende komponentbilder, varje innehållande en del av den renderade geometrin, för att sammanställa färdiga bilder. På grund av antagandet om oberoende kan reflektioner inte renderas precist. En metod i *screen-space* för att i efterhand lägga till reflektioner genom att använda buffrar för geometri och ytegenskaper föreslås. Reflektioner bestäms genom att approximera *ray tracing.*

Metoden tillåter perfekt reflektiva ytor och suddiga ytor med kvalitet som är jämförbar med kommersiella renderare för optimala scener. Den misslyckas dock med att fånga reflektioner av föremål som inte är synliga från den virtuella kameran med undantag för fall då dessa föremål är del av en fördefinierad kringliggande geometri. Metoden kan hantera dynamiska scener, kamerainställningar och ytegenskaper, inklusive varierande BRDF, om den inte används i realtid.

På grund av krav på kvalitet, renderingstid och bristen på grafikkort föreslås en metod för att kalkylera och spara reflektionsträffar som bygger på ett antagande om statisk geometri. Uppsnabbningstekniken tillåter godtyckliga variationer i ljussättning, texturer och reflektionsstyrka samt begränsade variationer i *Bump Mapping* och klarhet. Med sparade reflektionsträffar uppnås acceptabla tider för den avsedda applikationen.

3 Acknowledgements

I would like to thank my examiner Ulf Assarsson at Chalmers University of Technology.

I would also like to thank Spark Vision for giving me the opportunity to work on my master thesis. In particular, I would like to thank my supervisors Johan Nilsson and Arash Ohadi as well as Malin Bjelle and Herman Carlsson for their help with the company process and 3D Studio Max and V-Ray usage.

4 Table of Contents

5 Introduction

5.1 Background

Spark Vision produces product configurators that allow customers to customize products with different features and receive instant visual feedback. These customer requirements can then be used to generate cost estimates for the customer and product specifications for the companies. The visual feedback can, for example, be produced by individually pre-rendering every possible combination of features, through real-time 3D rendering or by composing layers of component images. This thesis work focuses on the latter approach. This approach is used at Spark Vision because real-time rendering does not meet the standards of photorealism needed and rendering every combination of features separately results in an exponentially growing number of images as the number of different features increases.

The component images contain parts of the final image and are rendered individually. Different component images are chosen, based on user input, and are layered to produce the final image. The purpose of this approach is to reduce the number of images that need to be rendered and stored by allowing the same component to be used in various different combinations.

A problem with combining component images is that global illumination effects such as reflections and color bleeding cannot be captured accurately as these effects are dependent on other components in the final image. Inaccurate or missing reflections in particular are very noticeable when using a product configurator for indoor tiles as the surface properties of such tiles result in strong reflections. To circumvent this, component images were previously either rendered with incorrect neutral, often greyscale, surrounding geometry which lessened the effect of the incorrect reflections or by using one specific set of surface properties for the surrounding environment resulting in correct reflections for that particular setup but incorrect reflections for all other setups.

5.2 Problem Statement and Purpose

While the problem originally arose from the need to render images component-wise in such a way that no pair of geometry and surface properties would need to be rendered more than once the problem being addressed is more general than that. The aim of this thesis work is to find a way to add reflections to a rendered image, given auxiliary data, in a way that captures the surface properties of the rendered image and is coherent with the techniques and parameters used when rendering the original image. The approach used is to add reflections as a post

processing effect by approximating ray tracing using screen space data and solve the following problems:

- Identify the auxiliary data needed
- Find a way to approximate the ray collision of ray tracing with screen space sampling
- Handling misses in screen space
- Handling glossy effects
- Identify and remove or alleviate artifacts
- Find applicable speed-up techniques

Maximum running time was never explicitly stated. Thus, acceptable running time is based on interpretation of the implicit statement of real-time with respect to a post-processor and set to 300ms.

5.3 Limitations

The engine used to combine and show the images is run on both web servers and as a standalone application on both PCs and handheld devices. Because of this no GPU implementation will be explored.

Evaluation of image quality will not be based on either user tests or any formal methods. Instead, images will be compared to ground truth images rendered using V-Ray.

There are several global illumination effects that cannot be accurately rendered when using component images. For this thesis work only reflections will be explored.

6 Theory and Previous Work

6.1 Reflections in real-time rendering

Planar reflections are rendered using a multi-pass approach. The first pass renders the scene from the camera, excluding the mirrored surface. This mirrored surface is rendered to create a stencil mask. In the second pass the scene is rendered again from the reflected view-point, drawing over the masked bits. Reflections of higher depth can be accomplished by rendering the scene from the transformed view-point of each reflector and recursively transforming the view-point and rendering again for each reflective surface visible until some given maximum depth is reached. [1]

This technique can be extended to approximate curved reflectors by rendering a distorted projection as seen from the transformed view-point. [2]

Planar reflections are computationally expensive because if more than one reflector is present the scene must be rendered once for each reflector and then again for each visible reflector in each step of the recursion. This is problematic for scenes with many polygons. Furthermore, it requires full knowledge of the scene primitives being rendered and thus cannot be used as a post-processing technique. [3]

Another approach to real-time reflections is environment mapping. Environment mapping was first proposed by Blinn and Newell in 1976. They use a sphere map around the reflective object onto which the surrounding scene is projected using a single parameterization. This sphere map is then sampled during rendering. A common alternative is to use cube maps in which the surrounding scene is projected onto one of six cube faces. [4] [5]

One problem with classic environment mapping as proposed by Blinn and Newell is that it is based on the assumption of an infinitely distant environment. When this assumption is broken the reflections are inaccurate. Furthermore, the assumption means that classic environment mapping cannot represent parallax. Classic environment mapping also fails to capture local reflections as the reflector itself is omitted during the construction of the environment map.

Several possible solutions have been proposed. One approach is to use several view-dependent radiance environment maps each of which is correct only for a single view-point. These radiance environment maps are then sampled depending on the view-point to achieve view-dependence and parallax. Similarly, multiple location and view-dependent environment maps can be used to create

Parameterized Environment Maps. Another proposed approach is to use 4D light fields in place of 2D environment maps where a light field can be viewed as 2D array of images. Each reflected ray is indexed based on its view-point. [6][7][8]

Environment mapping has also been extended to glossy reflections. One approach is to pre-filter the environment maps based on the surface's Bidirectional Reflectance Distribution Function (BRDF). The BRDF is sampled at a set of viewing directions to create lobes.These are used to filter the environment map resulting in a three-dimensional environment map where one dimension is the viewing direction. [9]

Another approach is to use *importance sampling*. Importance sampling uses the surface's BRDF to determine a set of sampling directions used to sample the environment map. The results of these samples are then summed to determine the final reflection color. In order to avoid having to handle random numbers on the GPU deterministic importance sampling is used instead which causes aliasing. To circumvent this, samples are filtered by sampling from a certain mipmap-level of the environment map depending on the probability of the sample. Importance sampling will be covered in detail in the next chapter. [10]

In order to generate an environment map full knowledge of the scene is required which makes approaches using environment mapping a poor alternative as a post-processing technique.

For this application both environment maps and planar reflections could be generated offline and stored. They could then be used after the layering of component images to add reflection. However, the number of environment maps or planar reflection maps needed to create accurate reflections would depend on the number of possible combinations of features which is what the thesis work is attempting to avoid. Thus, a better alternative would be the naïve approach of rendering every possible combination of features to begin with.

6.2 Reflections in ray tracing

The general recursive ray tracing algorithm used in classic ray tracing, outlined by Whitted in 1977, traces a ray for each pixel from the camera view-point. The closest collision is used to shade the pixel. From the collision point a shadow ray for each light source is traced to determine whether or not the surface is occluded as seen from that light source. For every un-occluded light source the surface properties are used to shade the pixel. If the surface is specular a reflection ray is traced recursively in the reflection direction. Thus, reflections are part of the rendering algorithm for ray tracing. A transmission (or refraction) ray is traced

recursively for refractive materials in the refraction direction determined by the index of refraction of the two media and Fresnel's law of refraction. Refraction will not be covered in any detail as it is not the focus of the thesis work. [11]

Classic ray tracing is good at capturing reflections of perfectly specular surfaces. However, it cannot properly render glossy reflections. To handle glossy reflections *distributed ray tracing* or *distribution ray tracing* is used instead. In distributed ray tracing several rays are traced per pixel, using some sub-sampling scheme. An important variant of distribution ray tracing is Monte-Carlo ray tracing. Monte-Carlo methods in general are computational algorithms that use repeated random sampling to reach a result. Basic Monte-Carlo *eye ray tracing*, i.e. ray tracing in which rays start from the eye or camera and propagate towards the light, can be summarized as follows:

1) Choose a ray
2) Find the closest point of intersection
3) Randomly choose either
    a) Emission
        i) Calculate emitted light times sample weight
    b) Reflection
        i) Randomly scatter the ray according to the BRDF with updated sample weight
        ii) Go to 2

[12]

The interesting part here is 3b in which the glossy reflection of the surface is determined. A naïve approach to this problem is to scatter the ray in a uniformly distributed random direction in the hemisphere and use the value of the BRDF as weight for the given incoming and outgoing directions. However, a better way is importance sampling, i.e. sampling in the directions where the BRDF is greater. If the probability distribution used to sample hemisphere correspond to the surface BRDF no weighing of samples is needed.

6.2.1 Importance sampling the Phong BRDF

The sampling direction is defined in a coordinate system in which the specular direction is the z-axis (0, 0, 1). The specular direction is the perfect reflection of the incoming direction about the surface normal. The sampling direction is defined using the two spherical coordinates θ and ϕ. θ is the angle between the specular direction and the sample direction and ϕ is the rotation of the sample direction about the specular direction.

The Phong BRDF assumes that the reflected light is based on a cosine falloff from the specular direction. Thus, the density d of the sampled reflected rays is defined as

$$d = \cos(\theta)^{n}$$

Where n is the shininess of the material and θ is the angle between the perfect specular reflection and the sample direction. In order to use this cosine lobe as a PDF (probability distribution function) p it needs to be normalized to integrate to 1 over the hemisphere. Since spherical coordinates are used the BRDF is written as

$$\cos(\theta)^{n} \sin(\theta)$$

Therefore the integral over the hemisphere is

$$\int_0^{\pi/2} \cos(\theta)^{n} \sin(\theta)\, d\theta = \left[\frac{-\cos(\theta)^{n+1}}{n+1}\right]_0^{\pi/2} = \frac{-\cos(\pi/2)^{n+1} + \cos(0)^{n+1}}{n+1}$$

$$= \frac{1}{n+1}$$

$$\int_0^{2\pi}\int_0^{\pi/2} \cos(\theta)^{n} \sin(\theta)\, d\theta d\phi = \int_0^{2\pi} \frac{1}{n+1} = \left[\frac{\phi}{n+1}\right]_0^{2\pi} = \frac{2\pi}{n+1}$$

Dividing the BRDF with this integral yields

$$p(\theta, \phi) = \frac{\cos(\theta)^{n} \sin(\theta)}{\int_0^{2\pi}\int_0^{\pi/2} \cos(\theta)^{n} \sin(\theta)\, d\theta d\phi} = \frac{(n+1)}{2\pi} \cos(\theta)^{n} \sin(\theta)$$

In order to sample the PDF it is transformed into a CDF (Cumulative Distribution Function) by integration over the angular range. This CDF is then inverted in order to map from uniform samples to direction angles. The two direction angles are first separated into different PDFs. The PDF of $\theta$is determined by marginalizing out ϕ by integrating over the whole range of ϕ

$$p(\theta) = \int_0^{2\pi} p(\theta, \phi) d\phi = (n+1) \cos(\theta)^{n} \sin(\theta)$$

The PDF of ϕ can now be defined as the conditional probability given the value of θ

$$p(\phi|\theta) = \frac{p(\theta, \phi)}{p(\theta)} = \frac{1}{2\pi}$$

Converting the two PDFs into CDFs and inverting them results in

$$P(\theta) = \int_0^{\theta} (n+1)\cos(\theta)^{\mathrm{n}}\sin(\theta) = -\cos(\theta)^{n+1} + \cos(0)^{\mathrm{n}+1} = 1 - \cos(\theta)^{n+1}$$

Defining a uniform random variable ξ between 0 and 1 and $P(\theta) = \xi$

$$\xi = 1 - \cos(\theta)^{\mathrm{n}+1} \rightarrow \cos^{-1}(\sqrt[n+1]{1-\xi}) = \theta$$

If ξ is random variable between 0 and 1 then so is $\xi_1 = 1 - \xi$ so the expression can be simplified to

$$\theta = \cos^{-1}(\sqrt[n+1]{\xi_1})$$

Analogously

$$\xi_2 = P(\phi) = \int_0^{\phi} \frac{1}{2\pi} = \frac{\phi}{2\pi}$$

$$\phi = 2\pi\xi_2$$

These results are fairly intuitive. Since the Phong BRDF is isotropic the rotation of the sample vector about the reflection vector is uniformly distributed as expected. Furthermore, the angle between the sample vector and the reflection vector depends only on the shininess of the surface. [10][12]

6.2.2 Importance sampling the Blinn BRDF

The Blinn BRDF is based on a microfacet distribution function, i.e. a distribution of normal vectors about the surface normal. Such BRDFs are the product of three terms; normal distribution (NDF), geometry and Fresnel reflectance which are divided by two cosine terms. Because the NDF accounts for most of the variation it can be sampled directly with good results.

The NDF of the Blinn BRDF is a cosine falloff from the surface normal. This cosine falloff is identical to the Phong BRDF and as such the previous derivation would be identical with the only difference being that the coordinate system has the normal vector as the z direction (0, 0, 1). Thus, rather than deriving a new sample reflection direction directly an importance sampled half vector is derived instead.

$$\theta_{\mathrm{H}} = \cos^{-1}(\sqrt[n+1]{\xi_1})$$

$$\phi_H = 2\pi\xi_2$$

Here θ_{H} is the angle between the half vector and the surface normal and ϕ_H is the rotation of the half vector about the surface normal. The sample direction is then determined by reflecting the incoming direction about the half vector. [14]

7 Analysis

7.1 Overview

7.1.1 Buffers

In order to trace reflections in screen-space some additional data is needed besides the original image. These data are stored in buffers output by the original renderer. For this thesis work these buffers are rendered using ray tracing and stored on the hard drive. However, the techniques outlined have no limitations in terms of the original renderer and should be applicable even to real-time GPU based renders such as game graphics engines. Depending on which techniques outlined are used some of these buffers are not needed. For sake of clarity all of them are listed below.

Table 1. Buffers

C_f	Color buffer for the front side of objects
C_b	Color buffer for the back side of objects
P_f	Position buffer for the front side of objects
P_b	Position buffer for the back side of objects
N_f	Normal buffer for the front side of objects
N_b	Normal buffer for the back side of objects
F_f	Reflection Filter buffer for the front side of objects
F_b	Reflection Filter buffer for the back side of objects
R	Reflection Vector buffer for the initial bounce of mirror reflectors
S_f	Shininess buffer for the front side of objects
S_b	Shininess buffer for the back side of objects
C_{out}	Not an input buffer, the color output by the post-processor
R_{raw}	Not an input buffer, the raw reflection output by the post-processor

The renderer used for this thesis work is not able to output shininess. Thus, the images output from the post-processor use either uniform shininess or shininess is set arbitrarily over the image. The buffers are listed here despite this for clarity. For example buffers see Appendix A.

7.1.2 Strength of reflections

There are many different ways of determining the strength of reflections based on surface properties and the angle between of incoming ray and the normal vector of the surface. For this thesis work different surfaces use different techniques for determining this strength. Some surfaces use the common Fresnel reflection

technique while for others the strength is user defined, either by a value independent on the angle or by using an ad hoc curve. Therefore, different techniques are not covered in detail as regardless of the underlying technique each pixel in the image will correspond to one set of surface properties with one incoming angle. Thus, each pixel will have a corresponding number between 0 and 1 which is a factor determining the strength of the reflection. These values are stored in the reflection filter buffers F_f and F_b. The difference between these two buffers is detailed in the chapter on screen-space sampling. Thus the final color of a pixel (x, y) is determined by

$$C_{out\,(x,y)} = min(1, C_{f\,(x,y)} + c_{r_{(x,y)}} F_{f\,(x,y)})$$

where $c_{r_{(x,y)}}$ is the color of the raw reflection of the pixel and $C_{f\,(x,y)}$ is the color of the front color buffer. The reflection filter is thus considered an additional input to the post processor and no calculations of these values are done during the post processing. Thus, the problem is reduced to determining the value of c_r for each pixel given a set of surface properties.

Since the strength of reflections usually depends on the angle of the incoming ray it is incorrect to use the same buffer for secondary bounces as these values are output given the incoming angle as seen from the camera. Despite this, these values are used as an approximation for this thesis work. A more general approach would be to determine these values in the post-processor, requiring additional buffers for IOR values, curves and constant values. This will not be explored.

7.1.3 Original render settings

When creating the buffers not all render settings will give good results. In particular, heavy use of anti-aliasing will result in the color of certain pixels to be averaged from different surfaces. This is noticeable as the position and normal vector of the pixel needs to be determined from only one surface. Thus, if anti-aliasing is used before the reflection post-processing these averaged colors will show in the reflections when the color of only one of the surfaces should be used. It is therefore best to postpone anti-aliasing until after the post-processor is used or to not use anti-aliasing.

7.2 Screen-space sampling

As described in the overview the strength of the reflection of a pixel is stored in the buffer F_f. Thus, on the right hand side of the equation

$$C_{out\,(x,y)} = min(1, C_{f\,(x,y)} + c_{r_{(x,y)}} F_{f\,(x,y)})$$

only the raw reflection c_r is unknown. To determine the value of c_r from screen-space data the normal and position of the pixel, stored in $N_{f_{(x,y)}}$ and $P_{f_{(x,y)}}$ respectively, are used to determine its reflection vector $\boldsymbol{r}_{(x,y)}$. Both normal and position are given in view-space, i.e. a coordinate system in which the camera is situated at the origin looking in the z-direction. For the purpose of this text, the camera is assumed to be oriented to look in the negative z-direction. Thus, the incidence vector of the pixel (x, y) from the camera is equal to its position in view-space.

$$\boldsymbol{i}_{(x,y)} = P_{f_{(x,y)}} - \boldsymbol{p_c} = P_{f_{(x,y)}}, \boldsymbol{p_c} = (0,0,0)$$

where $\boldsymbol{i}_{(x,y)}$ is the incidence vector and $\boldsymbol{p_c}$ is the position of the camera.The following formula is then used to calculate the reflection vector:

$$\boldsymbol{r} = \frac{(\boldsymbol{i} - 2(\boldsymbol{n} * \boldsymbol{i}) * \boldsymbol{n})}{|(\boldsymbol{i} - 2(\boldsymbol{n} * \boldsymbol{i}) * \boldsymbol{n})|}$$

$$\boldsymbol{r}_{(x,y)} = \frac{(\boldsymbol{P_{f_{(x,y)}}} - \boldsymbol{2}\left(N_{f_{(x,y)}} * \boldsymbol{P_{f_{(x,y)}}}\right) N_{f_{(x,y)}})}{|(\boldsymbol{P_{f_{(x,y)}}} - \boldsymbol{2}\left(N_{f_{(x,y)}} * \boldsymbol{P_{f_{(x,y)}}}\right) N_{f_{(x,y)}})|}$$

Alternatively, the reflection vector itself can be output by the original renderer and stored in the buffer R, in which case $\boldsymbol{r}_{(x,y)} = R_{(x,y)}$. Note that this buffer is only relevant for the first bounce. To avoid self reflections the position should be offset by some ϵ in the normal direction.

$$\boldsymbol{p}_{view} = P_{f_{(x,y)}} + \epsilon N_{f_{(x,y)}}$$

A sample position in view-space $\boldsymbol{p}_{\boldsymbol{s}_{(x,y)}}$ for the pixel (x, y) following the reflection vector $\boldsymbol{r}_{(x,y)}$ from view-space position $\boldsymbol{p}_{view}$ is thus a position adhering to the following equation:

$$\boldsymbol{p}_{\boldsymbol{s}_{(x,y)}} = \boldsymbol{p}_{view} + \boldsymbol{r}_{(x,y)} * t, t > 0$$

where t is a real number. t is then iteratively increased linearly by some delta $d > 0$. The choice of d depends on the scene being rendered. Every sample position $\boldsymbol{p_s}$ is then tested for collision.

7.2.1 Collision testing

In order to test for collision the projection parameters used to render the original image are needed. These parameters are used to project the sampled view-space

position $\boldsymbol{p_s}$ to screen-space. This projection is for the purpose of this chapter considered a function

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

which maps a view-space position to a screen-space texture coordinate. The screen-space position given by the *project* function is thus where the sampled screen-space position would have ended up in the final image were it rendered. The details of the projections used are covered in Appendix B.

In order to un-project, i.e. go from screen-space coordinates to view-space position the depth value, i.e. the projected Z-axis, of the pixel is also needed which is not known. However, since the view-space position of the closest rendered object for each pixel is stored in the front position buffer P_f un-projection is simplified to a texture look-up.

For every sample view-space position $\boldsymbol{p_s}$ the following values can be calculated:

$$\boldsymbol{coord_s} = project(\boldsymbol{p_s})$$

$$\boldsymbol{p_{front}} = P_{f_{\boldsymbol{coord_s}}}$$

To test for collision all that is left to do is to check if p_{front} is in front of $\boldsymbol{p_s}$ as seen from the camera.

$$\boldsymbol{p_s}.z < \boldsymbol{p_{front}}.z$$

The raw reflection color $c_{r_{(x,y)}}$ is then the color of C_f at coordinate $\boldsymbol{coord_s}$.

$$c_{r_{(x,y)}} = C_{f_{\boldsymbol{coord_s}}}$$

This approach works under the assumption that each pixel extends infinitely behind the area it covers, as seen from the camera, as it does not take into account the distance between $\boldsymbol{p_s}$ and $\boldsymbol{p_{front}}$ nor the angle between $\boldsymbol{r}_{(x,y)}$ and $N_{f_{coor\ d_s}}$. Thus, a ray entirely behind an object will still be considered as colliding with that object, leading to incorrect reflections. An ad hoc solution is to assign a global *thickness* parameter which is then used for collision testing. The comparison

$$\boldsymbol{p_s}.z < \boldsymbol{p_{front}}.z$$

would then instead be

$$\boldsymbol{p_s}.z < \boldsymbol{p_{front}}.z \wedge \boldsymbol{p_s}.z > \boldsymbol{p_{front}}.z - thickness$$

While an improvement, there is no way of defining a global *thickness* parameter in such a way that it would not lead to artifacts for some surfaces.

Another problem is that a ray considered colliding, even correctly, with the back side of an object would still return the color value of the front side of the object.

A solution to both these problems is to render the entire scene twice. Once using the original render settings and once with front face culling enabled and back face culling disabled. Another approach, if front face culling is not available to the renderer, is to invert all the normal vectors in the scene and render with back face culling enabled. Both these approaches will result in the colors, view space positions, normal vectors, reflection filter and (potentially) shininess of the back side of the closest objects. These are stored in the buffers C_b, P_b, N_b, F_b and S_b respectively.

With these buffers, in addition to buffers of the original rendering settings, the comparison can then be replaced with

$$|\boldsymbol{p_s}.z| > |\boldsymbol{p_{front}}.z| \wedge |\boldsymbol{p_s}.z| < |\boldsymbol{p_{back}}.z|$$

where

$$\boldsymbol{p_{back}} = P_{b_{\boldsymbol{coord_s}}}$$

To determine whether $\boldsymbol{r_{(x,y)}}$ collides with the back side of the object the relative distance from $\boldsymbol{p_s}$ to $\boldsymbol{p_{front}}$ and $\boldsymbol{p_{back}}$ are used together with the angle between $\boldsymbol{r_{(x,y)}}$ and $N_{b_{coor\ d_s}}$. Thus, $\boldsymbol{r_{(x,y)}}$ is considered colliding with the back side if

$$|\boldsymbol{p_s}.z - \boldsymbol{p_{front}}.z| > |\boldsymbol{p_s}.z - \boldsymbol{p_{back}}.z|$$

and

$$-\boldsymbol{r}_{(x,y)} * N_{b_{coor\ d_s}} > 0$$

The raw reflection color $c_{r_{(x,y)}}$ is thus

$$c_{r_{(x,y)}} = C_{b_{\boldsymbol{coord_s}}}, |\boldsymbol{p_s}.z - \boldsymbol{p_{front}}.z| > |\boldsymbol{p_s}.z - \boldsymbol{p_{back}}.z| \wedge -\boldsymbol{r}_{(x,y)} * N_{b_{coor\ d_s}} > 0$$

$$c_{r_{(x,y)}} = C_{f_{\boldsymbol{coord_s}}}, Otherwise$$

Colloquially, a ray hits the back side of an object if it is closer to the back side and collision with the back side is possible given its normal. The sampling stops when the projected coordinates $coord_s$ is outside of the image or when some threshold

has been reached in terms of the number of samples. A different approach is outlined in the next chapter.

7.3 Handling misses in screen-space

A problem with any screen-space solution is that only screen-space data exists. In the case of reflections some reflected rays will not hit any of the pixels in the original image. These rays could be ignored, but that leads to abrupt changes between hits and misses in screen-space. In order to handle this problem a solution based on a simplifying assumption is proposed. The assumption being that the scene being rendered is bounded by a box. This assumption might seem naïve, but in the case of the application in question the scenes being rendered are indoor scenes with a single room. Thus, the bounding box is the walls, floor and ceiling of said room. Using this assumption a representation of the scene's boundary can be created and used for rays that miss in screen-space. Thus, classic ray tracing using a significantly simplified scene takes over when screen-space sampling fails.

In order to determine whether or not a pixel is part of a scene boundary the coordinates of the corners of these boundaries must be known. These coordinates could be given in view-space directly, thus requiring no additional input. To simplify the extraction of these values however, they are instead given in world-space as for the scenes in this application, the boundary box forms an axis-aligned box in world-space and can thus be represented using six real numbers: min and max for each axis. With the boundary coordinates given in world-space they need to be transformed to view-space thus requiring the camera position, camera target and camera up-vector in order to create a view-matrix. View-matrices are covered in detail in plenty of computer graphics resources and will not be covered again here.

7.3.1 Ray-quad intersection test

With a quad for each boundary defined in view-space a way to determine where a ray collides with the quad is needed. Ray-quad intersection testing is done by an adaptation of the ray-triangle method presented in [16]. For this ray-triangle intersection test method a triangle is represented by three points: V_0, V_1 and V_2 and a point on the triangle is defined as

$$P(u, v) = (1 - u - v)V_0 + uV_1 + vV_2$$

$$u \geq 0, v \geq 0$$

$$u + v \leq 1$$

where V_0, V_1 and V_2 are the three corners of the triangle and u and v are the two-dimensional texture coordinates. A ray is defined as a starting point $\boldsymbol{o}$ and a direction vector $\boldsymbol{d}$. Testing a ray against a triangle results in the distance t and the texture coordinates u and v. Thus, the test can be used for texturing. To adapt this solution to quads the definition is replaced with

$$P(u, v) = (1 - u - v)V_0 + uV_1 + vV_2$$

$$u \geq 0 \leq 1, v \geq 0 \leq 1$$

Where V_0, V_1 and V_2 form a right triangle and $V_1 + V_2 - V_0 = V_3$ with V_3 being the fourth point in the quad. The normal of the boundary can then be calculated as the cross product of $V_1 - V_0$ and $V_2 - V_0$.

7.3.2 Creating textures for boundaries

Both this chapter and the chapter on *Texture Completion* are based on the idea of creating the boundary textures from screen-space data. This is not necessary for the technique of using screen-space boundaries in and of itself. Using pre-rendered textures instead might prove to give more accurate results. Furthermore, the techniques outlined are tailored for the given application and may not work for surfaces that do not conform to the assumptions stated. An overview of the technique is outlined despite this for completeness.

For each boundary a texture can be created and filled. Consider a single boundary quad with points V_0, V_1 and V_2, normal N_Q and texture T as well as a pixel (x, y) in the original image with color $C_{f_{(x,y)}}$, normal $N_{f_{(x,y)}}$ and view-space position $P_{f_{(x,y)}}$. $T_{u_0 v_0} = C_{f_{(x,y)}}$ if

- A ray-quad collision test with the ray **o** = $P_{f_{(x,y)}}$, **d** =$N_{f_{(x,y)}}$ and the quad associated with T with corners V_0, V_1 and V_2 results in u = u_0 and v = v_0
- $t < distance\_threshold$
- $\cos^{-1}(N_{f_{(x,y)}} * N_Q) < angle\_threshold$

Iterating over the pixels in the original image and testing them against all six boundaries will result in an orthogonal projection of each boundary. The values of the thresholds depend on the scene and projection parameters.

Some boundaries will, ignoring cases with extreme values of field of view, not have any pixels projected to their respective texture as they are not visible from the camera. For such boundaries some assumption needs to be made as there is no data available. For the images presented in this thesis work they are assumed

to be completely non-reflective and black, thus not contributing at all to the raw reflection.

7.3.3 Texture Completion

After creating the orthogonal projections of the boundaries the corresponding textures still contain only the data present in the original front color buffer. Before using the boundary textures they must be completed. The texture completion used applies Canny Edge Detection to each projection and then uses the heuristic outlined in [15] to determine the horizontal and vertical shift, h_s and v_s respectively. Using only vertical and horizontal shift is based on the assumption that significant tiles of indoor scenes are vertically and horizontally aligned. These values are used to determine, for each empty pixel, which pixel to use when filling said empty pixel.

Canny Edge Detection usually applies Gaussian blur to the image before finding the gradients. This turned out to achieve worse results for this application when applying the heuristic. Instead, Gaussian blur is only applied to empty pixels surrounded by non-empty pixels to improve edge detection. Empty pixel to which Gaussian blur has been applied are still filled like any other. Canny Edge Detection is outlined in numerous resources and will not be outlined again here.

Texture completion is done using the following steps:

- Apply Canny Edge Detection
    - Apply Gaussian blur to empty pixels
    - Find the gradients
    - Apply non-maximum suppression, ignoring diagonal edges
- Approximate h_s and v_s using the heuristic
- Fill the unknown pixels with respect to the values identified
- Pixels left unfilled are taken from the closest horizontal or vertical pixel

This technique works well for some surfaces and fails for others, usually if the structure of the surface lacks pattern, the lighting is significantly different in different areas or where some noise in the image is difficult to separate from actual pattern. The following images show an example in which the wall to the right is projected and completed.

Figure 1: *Example of a front color buffer.*

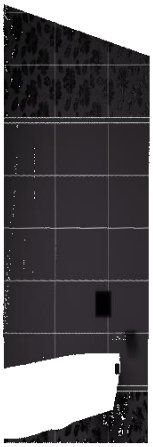

Figure 2: *Orthogonal Projection of one scene boundary based on the front color buffer.*

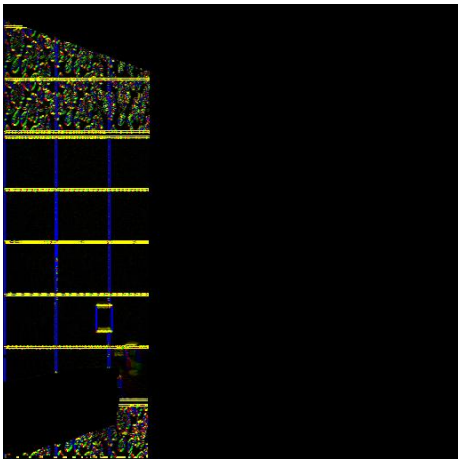

Figure 3: *Gradients acquired from an orthogonal projection using Canny Edge Detection. Yellow means horizontal, blue means vertical and red and green means diagonal depending on direction.*

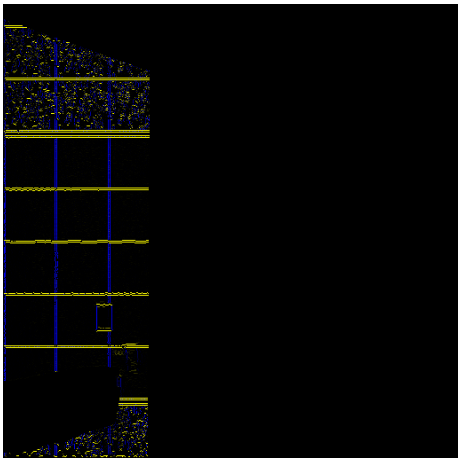

Figure 4: *Gradients remaining from an orthogonal projection using Canny Edge Detection after non-maximum suppression and ignoring diagonal edges.*

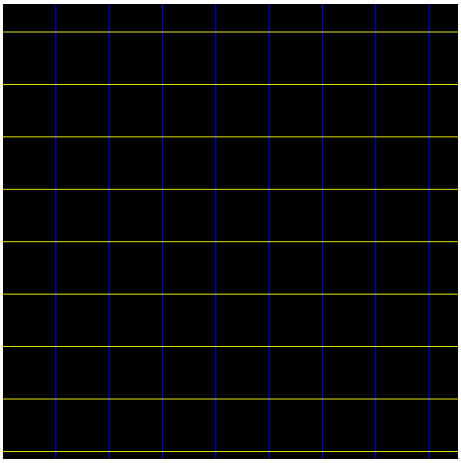

Figure 5: *The structure determined based on the random heuristic.*

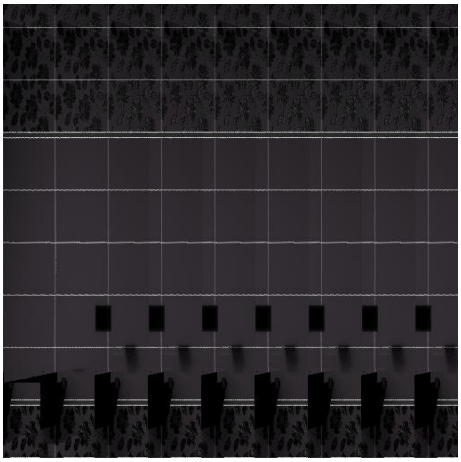

Figure 6: *The orthogonal projection filled based on the structure determined.*

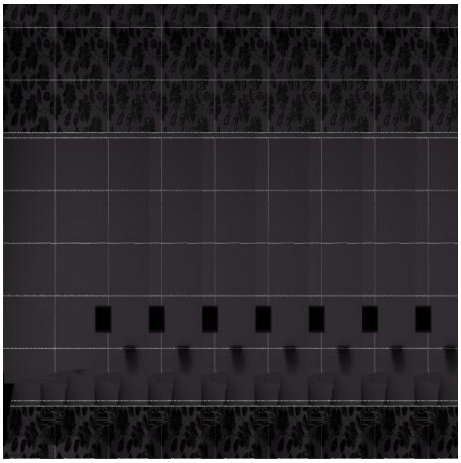

Figure 7: *The orthogonal projection filled again using the closest horizontal pixel.*

7.3.4 Using the boundary textures

Boundary textures can be sampled using the same ray-quad intersection test outlined previously. When screen-space sampling fails the ray is instead tested against the six boundaries, resulting again in *u* and *v* values. These values are used to perform a texture lookup in the boundary texture that was hit.

Given that the view-space limits of the scene are known when using this technique, the screen-space sampling can stop when either the projected coordinates $coord_s$ are outside of the image or when the sampled view-space coordinates $\boldsymbol{p_s}$ are outside the scene.

7.4 Achieving higher reflection depth

To achieve higher reflection depth a new origin $\boldsymbol{o}_{new}$ and direction vector $\boldsymbol{d}_{new}$ of a new reflection ray are computed from the previous origin $\boldsymbol{o}_{prev}$ and previous direction $\boldsymbol{d_{prev}}$. $\boldsymbol{d}_{new}$ can be computed using the same formula used for the initial reflection direction vector

$$\boldsymbol{d}_{new} = \boldsymbol{r} = \frac{(\boldsymbol{i} - 2(\boldsymbol{n} * \boldsymbol{i}) * \boldsymbol{n})}{|(\boldsymbol{i} - 2(\boldsymbol{n} * \boldsymbol{i}) * \boldsymbol{n})|}$$

where $\boldsymbol{n}$ is the normal of the collision position. For hits in screen-space at screen-space position (x, y) the normal of $N_{f_{(x,y)}}$ or $N_{b_{(x,y)}}$ is used depending on whether or not the $\boldsymbol{d_{prev}}$ hits the front side or back side of the object. For misses in screen-space the normal of the quad hit N_Q is used.

The new origin $\boldsymbol{o}_{new}$ in screen-space can be determined by $P_{f_{(x,y)}}$ or $P_{b_{(x,y)}}$ directly. However, because a pixel is not a single point in space but rather an area this position, $\boldsymbol{p}_{buffer}$ **,** does not necessarily adhere to the equation

$$\boldsymbol{p}_{buffer} = \boldsymbol{o}_{prev} + \boldsymbol{d}_{prev} * t\,, t > 0$$

Thus, using a position from the buffers directly will skew the reflections and can result in self-reflections. Therefore, a ray-plane intersection test is performed to determine $\boldsymbol{o}_{new}$**.** A plane is defined by the position and the normal of the relevant buffers, again depending on whether or not the back side or front side is hit. With the position $\boldsymbol{p}_{buffer}$ and the normal $\boldsymbol{n}_{buffer}$ from the buffers a plane is defined by

$$\boldsymbol{n}_{buffer} * (\boldsymbol{p}_{plane} - \boldsymbol{p}_{buffer}) = 0$$

Substituting $\boldsymbol{p}_{plane}$ with $\boldsymbol{o}_{prev} + \boldsymbol{d}_{prev} * t$ and solving for *t* yields

$$t = \frac{(\boldsymbol{p}_{buffer} - \boldsymbol{d}_{prev}) * \boldsymbol{n}_{buffer}}{\boldsymbol{d}_{prev} * \boldsymbol{n}_{buffer}}$$

And the new origin can then be computed by

$$\boldsymbol{o}_{new} = \boldsymbol{o}_{prev} + \boldsymbol{d}_{prev} * t$$

For misses in screen-space the ray-quad intersection test determines the distance *t* and the same formula can be used directly.

While a large improvement over using the buffers directly, this still results in self reflections for both hits and misses in screen-space. This is partly due to rounding errors and to the incorrect representation of pixels as points. To circumvent this $\boldsymbol{o}_{new}$ is offset in the normal direction by some ϵ.

$$\boldsymbol{o}_{new} = (\boldsymbol{o}_{prev} + \boldsymbol{d}_{prev} * t) + \boldsymbol{n} * \epsilon$$

If the ray tracing algorithm is defined in terms of view-space coordinates both hits and misses can be treated identically by recursion until a diffuse surface is hit, until some maximum depth is reached or until the cumulative weight, determined by the reflection filter along the recursion, reaches some threshold.

7.5 Glossy Effects

7.5.1 Importance Sampling

The details of importance sampling of the Phong and Blinn BRDFs are covered in the chapter *Theory and Previous Work*. Adapting these results to this thesis in order to determine the raw reflection color $c_{r_{(x,y)}}$ of a pixel (x, y) is done by determining a set of sampling directions based on the BRDF and shininess of the surface.

A single sampling direction $\boldsymbol{d}_s$ is determined by generating two uniformly distributed random variables ξ_1 and ξ_2 between 0 and 1. For Phong

$$\theta = \cos^{-1}(\sqrt[n+1]{\xi_1})$$

$$\phi = 2\pi\xi_2$$

Where

$$n = S_{f_{(x,y)}}$$

yields the sampling direction in spherical coordinates in specular space i.e. where the reflection vector $\boldsymbol{r}_{spec\ (x,y)}$ is the Z axis. $r_{spec\ (x,y)}$ is either determined by reflecting $P_{f\ (x,y)}$ about $N_{f\ (x,y)}$ or retrieved from $R_{(x,y)}$ directly. The Cartesian coordinates in specular space $\boldsymbol{d}_{spec} = (x_s, y_s, z_s)$ is given by

$$x_s = \cos(\phi_s)\sin(\theta_s)$$
$$y_s = \sin(\phi_s)\sin(\theta_s)$$
$$z_s = \cos(\theta_s)$$

which is transformed to view-space by

$$\boldsymbol{d}_s = x_s * \boldsymbol{u} + x_y * \boldsymbol{v} + x_z * \boldsymbol{w}$$

where

$$\boldsymbol{w} = \boldsymbol{r}_{spec\ (x,y)}$$
$$\boldsymbol{u} = \frac{\boldsymbol{a} \times \boldsymbol{w}}{|\boldsymbol{a} \times \boldsymbol{w}|}$$
$$\boldsymbol{v} = \boldsymbol{u} \times \boldsymbol{w}$$

and $\boldsymbol{a}$ is an arbitrary vector. An arbitrary vector can be used because only the specular direction defines the space. [10][13]

For Blinn

$$\theta = \cos^{-1}(\sqrt[n+1]{\xi_1})$$

$$\phi = 2\pi\xi_2$$

where

$$n = S_{f\ ((x,y))}$$

yields the half vector $\boldsymbol{h}_{norm\ (x,y)}$ in normal space i.e. where the normal vector $N_{f\ (x,y)}$ is the Z axis. The half vector in Cartesian view-space $\boldsymbol{h}_{(x,y)}$ is determined in the same way as the reflection vector for Phong except $\boldsymbol{w} = N_{f\ (x,y)}$. $\boldsymbol{d}_s$ is determined by reflecting $P_{f\ (x,y)}$ about $\boldsymbol{h}_{(x,y)}$. [14]

Using this sampling scheme will result in a set of sampling directions. Each of these directions is then traced using the same algorithm detailed above and $c_{r_{(x,y)}}$ is determined by calculating the mean of these samples. When using glossy reflections of higher reflection depth secondary bounces do not generate

additional rays; one ray is traced for each original sample. Furthermore, the rays should be skewed again for each bounce, based on the BRDF and shininess of the surface hit.

7.5.2 Adaptive blur

For a sufficiently large number of samples the importance sampling scheme converges to accurate results. However, in order to keep computation time down the number of samples for this thesis can typically not be greater than 8, even when using the speed-up pre-calculation detailed in the next chapter. With few samples the raw reflection suffers from sampling artifacts. To alleviate this issue the raw reflection is first rendered into its own buffer R_{raw} which is then blurred.

Standard techniques for blur such as Gaussian blur are not applicable as the size of the filter is static. Thus, the filter will either blur too much in places where the sampling spread is small or blur too little in places where the sampling spread is large.

To handle this, the average distance traveled by the sampling rays and the shininess of the pixel are used to determine the size of the filter for each pixel. This blurring is not physically accurate and a larger sample size is to be preferred when computation times allow for it. The filter is square and the size of the side the filter of pixel (x, y) is

$$h_{(x,y)} = w_{(x,y)} = \max(5, 2\,\mathrm{d}_{(\mathrm{x,y})}\tan(\cos^{-1}(0.5^{\frac{1}{1+\mathrm{S_{f(x,y)}}}})))$$

Where $\mathrm{d}_{(\mathrm{x,y})}$ average distance travelled by each ray of the pixel (x,y). The justification for the formula is that the expected value of a uniformly distributed random value between 0 and 1 is 0.5. The average angle of the samples $\cos^{-1}(0.5^{\frac{1}{1+\mathrm{S_{f(x,y)}}}})$ is used to estimate the spread of the samples. Note that this is an ad hoc solution as the distance is in view-space. However, it gives a good approximation and most importantly, lessens blurring of pixels with little sampling spread.

The main purpose of applying adaptive blur to the raw reflection is to alleviate sampling artifacts while not blurring areas with little to no spread. Thus, a simpler formula can be used for a particular scene.

$$h_{(x,y)} = w_{(x,y)} = \max(5, c\,\frac{d_{(x,y)}}{S_{f\,(x,y)}})$$

where c is a scene specific constant. While not based on any physical law it gives similar and faster results if c is tuned to accommodate the scene.

7.6 Speed-up techniques

It quickly became apparent during the thesis work that performing all the steps of the post-processor for every image would not be feasible given the time constraint of 300ms, the quality demands and the lack of a GPU. Even ignoring misses in screen-space, thus not requiring texture completion and using only perfect reflectors, thus requiring only one reflection vector per pixel, images still took several seconds to process. In order to process quickly enough the number of sampled positions for each reflection vector needed to be reduced significantly resulting in larger sampling steps. This leads to rays missing thin objects entirely and patterns start to emerge. The effect of reduced sampling is shown below.

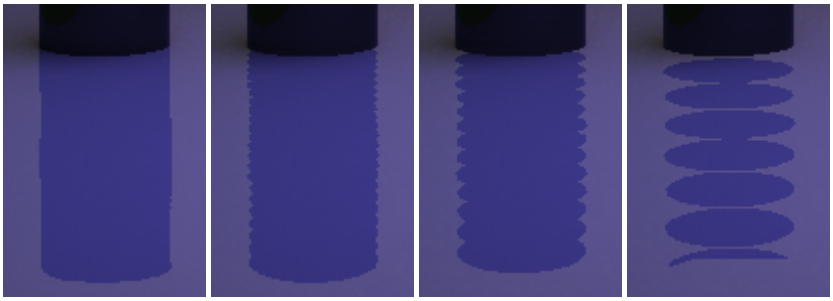

Figure 8: *The effect of reduced sampling of reflection rays.*

7.6.1 Reflection Caching

In order to accomplish the quality required while maintaining acceptable computation time the traced rays can be pre-computed, which simplifies the computations needed in real-time to mostly texture lookups and array indexing. The caching is based on a simplifying assumption, namely that the geometry of the scene does not change between frames. With this assumption the screen-space collision position of each sample can be stored and reused for the next frame. This assumption is far too restrictive for most applications. However, product configurators, the ones relevant for this thesis work in particular, generally only have one static scene with one static camera. What changes between frames is not the geometry but the surface properties. For scenes with a limited set of geometrical setups a cache for each of them can be pre-computed.

Geometry here refers to the positions of the pixels i.e. the buffers P_f and P_b. The reflection cache is created using all the buffers, including N_f, N_b, R, C_f, C_b, F_f, F_b, S_f and S_b. The rays are traced as before and the screen-space position hit is stored. Because the texture completion used for this thesis work matches every missing pixel of the boundary textures to exactly one pixel in screen-space the

caching is easily extended to handle misses in screen-space by treating misses as hitting the pixel used during completion. The data stored is as summarized below.

- For every pixel (x,y)
  - The average distance $d_{(x,y)}$ traveled by the samples
  - For every glossiness sample $g_{s_{(x,y)}}$
    - The direction vector of the sample $r_{(x,y)_s}$
    - The value of the PDF for the original shininess n_1 $p\left(\theta g_{s_{(x,y)}}\right) = (n_1 + 1)\cos(\theta g_{s_{(x,y)}})^{\mathrm{n}_1}$
    - For every recursion of secondary bounces
      - The sampled screen-space collision position $(x, y)_s$ of the traced ray
      - A Boolean $front_s$ signifying whether or not the front side of the object was hit

The reflection cache is used by iterating over all the pixels of the new buffers. The color of each glossiness sample is calculated by iterating over the secondary bounces and adding $C_{f\,(x,y)_s}$ or $C_{b\,(x,y)_s}$ depending on the value of $front_s$ weighed by the cumulative reflection filter, determined by $F_{f\,(x,y)_s}$ or $F_{b\,(x,y)_s}$, along the samples. $d_{(x,y)}$ is used for adaptive blur.

If there is no bump mapping and the shininess and BRDF of the surfaces remain constant then $r_{(x,y)_s}$ and $p(\theta g_{s_{(x,y)}})$ are not needed as the value of the PDF will be the same for each frame. The raw reflection color is then simply the average of the samples.

7.6.2 Glossy Reflections and Bump Mapping with Reflection Caching

Reflection caching can accurately be used for perfect non-glossy reflections. However, since perfect reflections are the result of infinite shininess with the sampling PDF reaching a delta function only samples in the perfect specular direction given the original bump map will be traced. Because of this no amount of weighing of the samples will approximate a different bump map as the weight of any sample not corresponding to the perfect reflection of a new bump map will be 0. This is also why tracing more than one ray for perfect reflections is superfluous as the result would be the same for each of them. Using reflection caching for perfect reflections must therefore be handled as a special case where one reflection cache for each possible bump map is created.

Glossy reflections sampled using the importance sampling scheme however will result in sampling rays that are relevant even for different bump maps and shininess levels. Using the additional data stored in the reflection cache the following formula is used to determine the raw reflection color of a pixel the pixel (x, y).

$$c_{r\,(x,y)} = \frac{\sum_{i=1}^{m} \frac{c_{s_i} p_2(r_{(x,y)_i})}{p_1(r_{(x,y)_i})}}{\sum_{i=1}^{m} \frac{p_2(r_{(x,y)_i})}{p_1(r_{(x,y)_i})}}$$

Where *m* is the number of samples, $r_{(x,y)_i}$ is the direction of sample i and c_{s_i} is the color of the sample retrieved by computing the weighted sum along the reflection depth.

Colloquially, the glossiness sample is weighed by the probability of the sample given the new shininess divided by the probability of the sample given the old shininess and summed. This sum is divided by the total weight of the samples.

The value of the PDF for the original normal vectors and shininess are given by reflection cache i.e. $p_1\left(r_{(x,y)_i}\right) = p\left(\theta g_{s_{(x,y)}}\right)$. For a new bump map and shininess value the computations depend on whether the surface uses Blinn or Phong. For Phong the probability of the new sample is given by

$$p_2\left(r_{(x,y)_i}\right) = (n_2 + 1)\cos(r_{(x,y)_s} * \mathrm{r}_{(\mathrm{x,y})_{spec}})^{\mathrm{n}_2}$$

Where

$$\mathrm{r}_{(\mathrm{x,y})_{spec}} = \frac{(P_{f\,(x,y)} - 2(N_{f\,(x,y)} * P_{f\,(x,y)}) * N_{f\,(x,y)})}{|(P_{f\,(x,y)} - 2(N_{f\,(x,y)} * P_{f\,(x,y)}) * N_{f\,(x,y)})|}$$

as before.

For Blinn, the probability is given by

$$p_2\left(r_{(x,y)_i}\right) = (n_2 + 1)\cos(N_{f\,(x,y)} * \boldsymbol{h}_{(x,y)})^{\mathrm{n}_2}$$

Where

$$\boldsymbol{h}_{(x,y)} = \frac{\mathrm{r}_{(\mathrm{x,y})_{spec}} - \frac{P_{f_{(x,y)}}}{|P_{f_{(x,y)}}|}}{|\mathrm{r}_{(\mathrm{x,y})_{spec}} - \frac{P_{f_{(x,y)}}}{\left|P_{f_{(x,y)}}\right|}|}$$

8 Results

The images shown are rendered using the buffers in Appendix A.

8.1 Image Quality

8.1.1 Perfect Reflections

These images are rendered using no gloss, i.e. infinite shininess using one sample per pixel which is independent of the underlying surface BRDF. The images to the left are rendered using the post-processor and the images to the right are rendered using V-Ray.

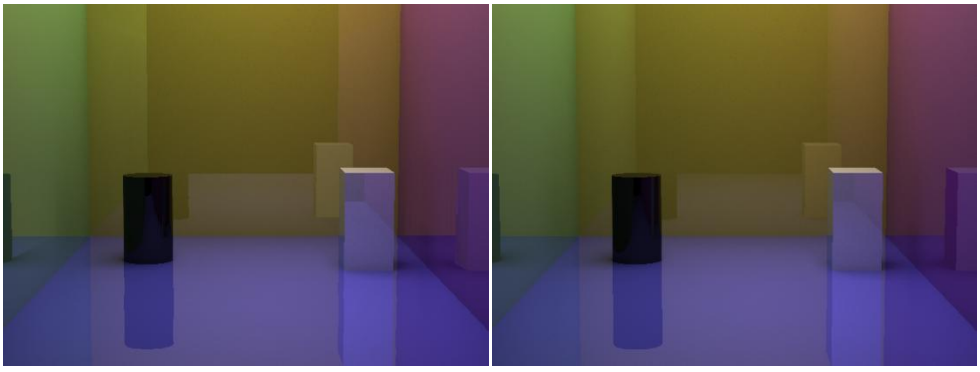

Figure 9: *Perfect reflections with 1 bounce using screen-space method (left image) and V-Ray (right image).*

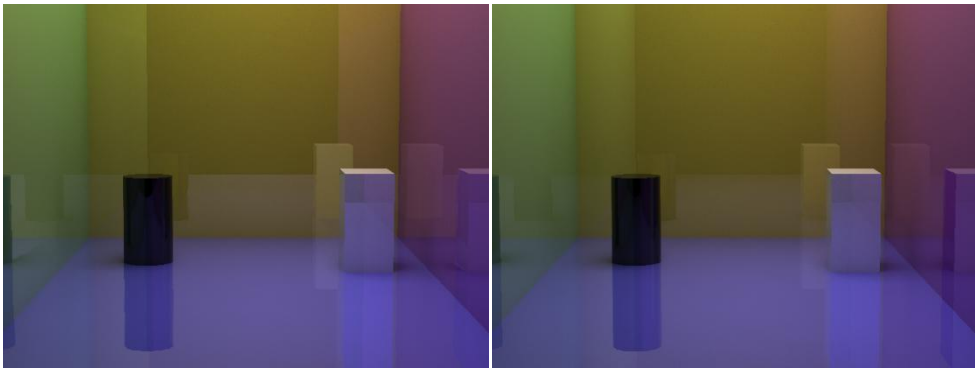

Figure 10: *Perfect reflections with 2 bounces using screen-space method (left image) and V-Ray (right image).*

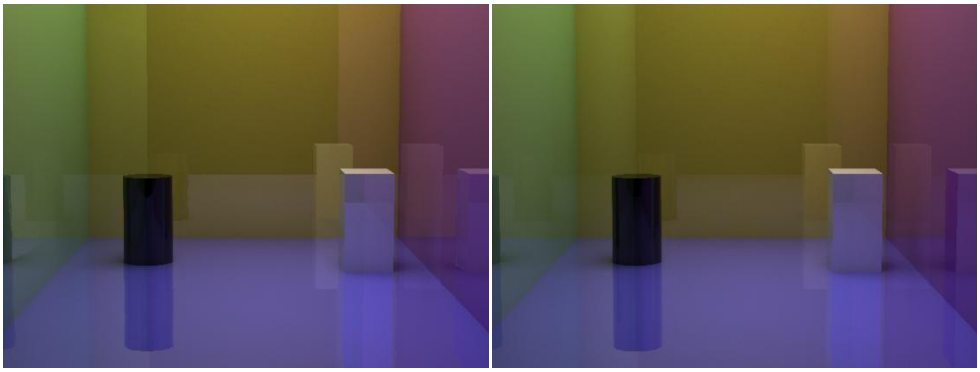

Figure 11: *Perfect reflections with 3 bounces using screen-space method (left image) and V-Ray (right image).*

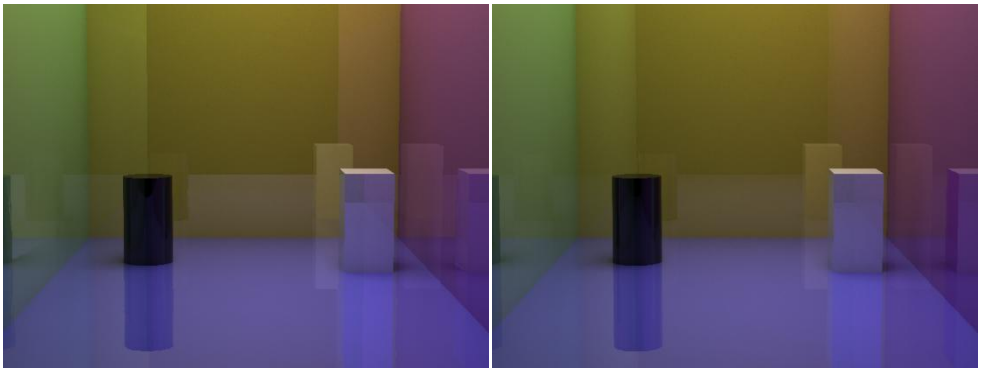

Figure 12: *Perfect reflections with 8 bounces using screen-space method (left image) and V-Ray (right image).*

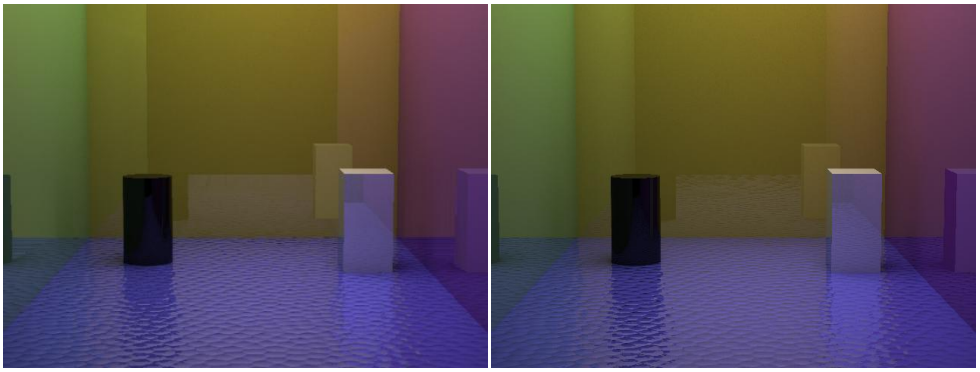

Figure 13: *Perfect reflections with 1 bounce and bump mapping using screen-space method (left image) and V-Ray (right image).*

These images can be considered proof of concept of the screen-space sampling scheme used. The difference in brightness is due to failures of the texture completion and incorrect lighting as the difference in incoming angle is ignored.

8.1.2 Glossy Reflections

These images are rendered using uniform BRDF and shininess values. All images are rendered using 8 samples, both for V-Ray and the post-processor. The images to the left are rendered using the post-processor and the images to the right are rendered using V-Ray.

8.1.2.1 Glossiness using Phong BRDF:

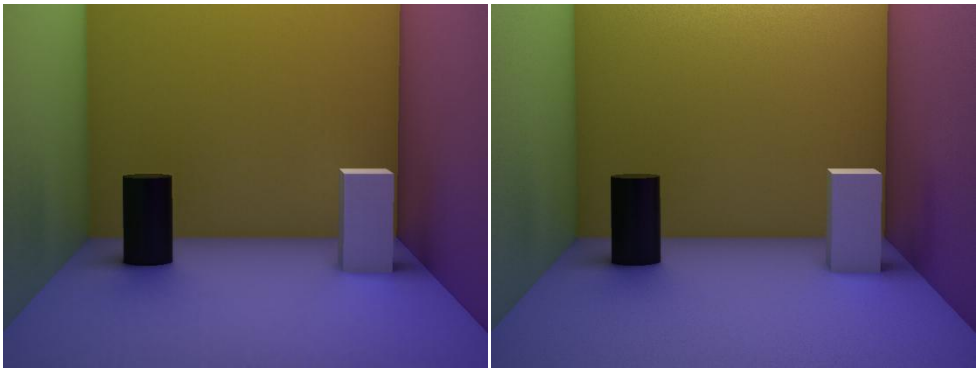

Figure 14: *Glossy reflections with 1 bounce using Phong BRDF with 8 samples and shininess 30 using screen-space method (left image) and V-Ray (right image).*

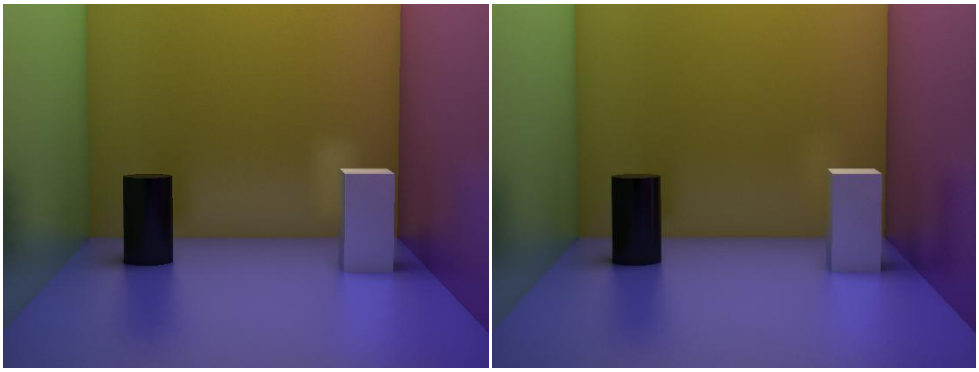

Figure 15: *Glossy reflections with 1 bounce using Phong BRDF with 8 samples and shininess 300 using screen-space method (left image) and V-Ray (right image).*

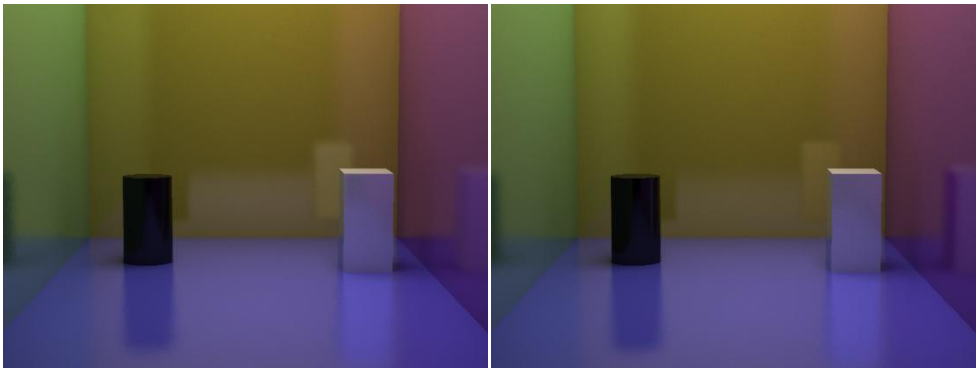

Figure 16: *Glossy reflections with 1 bounce using Phong BRDF with 8 samples and shininess 3000 using screen-space method (left image) and V-Ray (right image).*

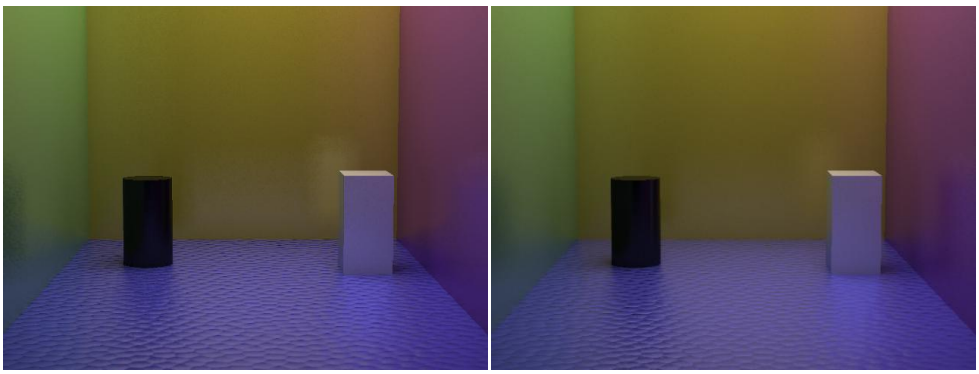

Figure 17: *Glossy reflections with 1 bounce and Bump Mapping using Phong BRDF with 8 samples and shininess 300 using screen-space method (left image) and V-Ray (right image).*

8.1.2.2 Glossiness using Blinn BRDF:

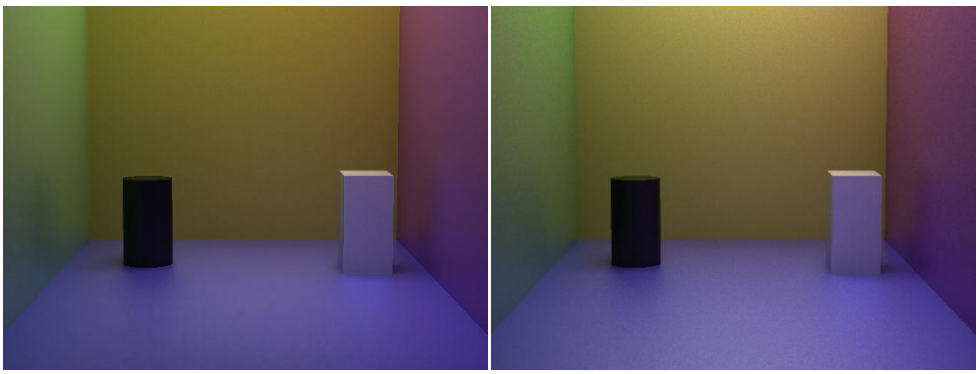

Figure 18: *Glossy reflections with 1 bounce using Blinn BRDF with 8 samples and shininess 30 using screen-space method (left image) and V-Ray (right image).*

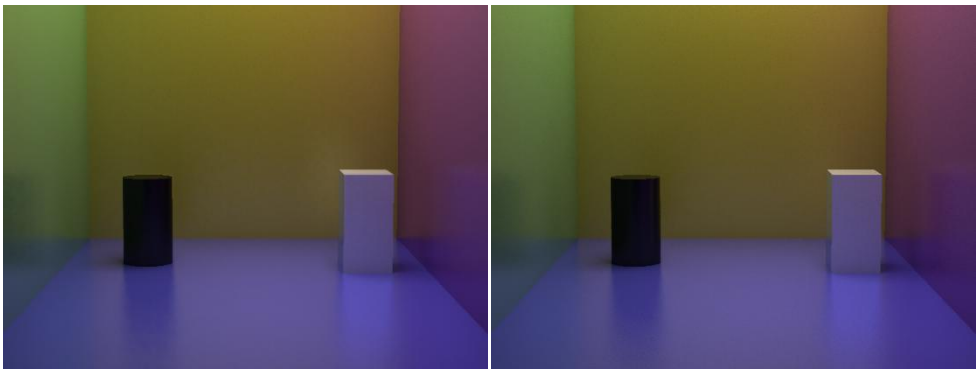

Figure 19: *Glossy reflections with 1 bounce using Blinn BRDF with 8 samples and shininess 300 using screen-space method (left image) and V-Ray (right image).*

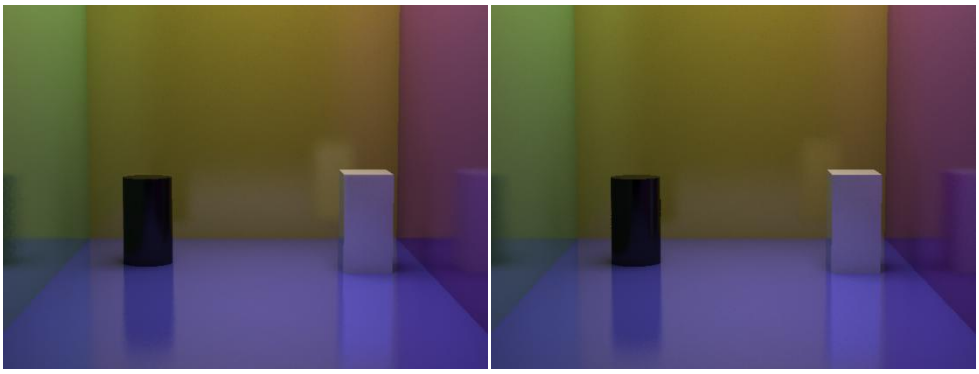

Figure 20: *Glossy reflections with 1 bounce using Blinn BRDF with 8 samples and shininess 3000 using screen-space method (left image) and V-Ray (right image).*

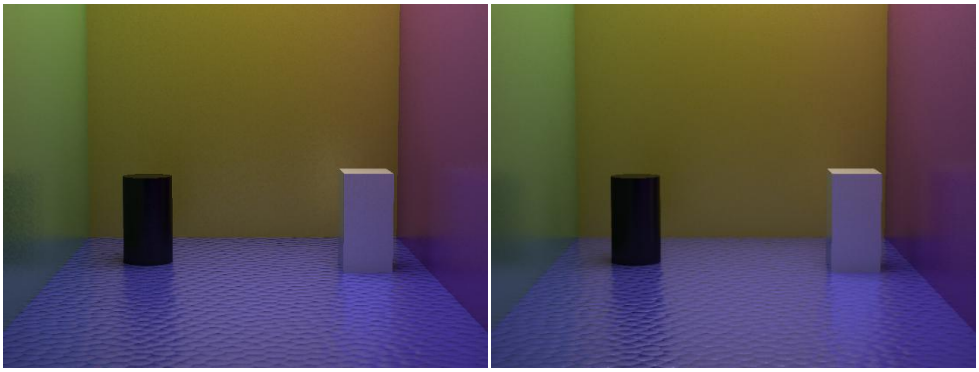

Figure 21: *Glossy reflections with 1 bounce and Bump Mapping using Blinn BRDF with 8 samples and shininess 300 using screen-space method (left image) and V-Ray (right image).*

8.1.3 Varying shininess and normal vectors for glossy reflections using reflection caching

The following images are the results of the weighted sampling used to allow for different glossiness values between frames. The images to the left are rendered using the post-processor by first creating the reflection cache using shininess 1000 and then rendered by weighing the samples according to the new shininess. The images to the right are rendered using the post-processor with correct shininess directly. All the images use Blinn BRDF.

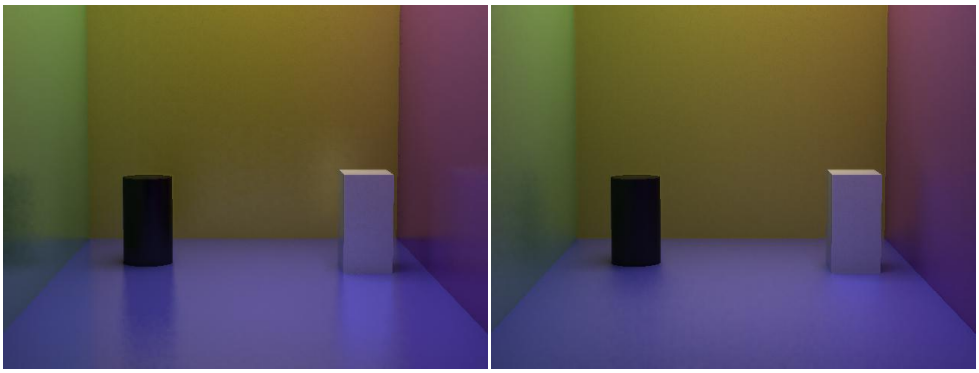

Figure 22: *Glossy reflections with 1 bounce using Blinn BRDF with 8 samples and shininess 30 using screen-space method with reflection cache created with shininess 1000 (left image) and screen-space method with correct shininess (right image).*

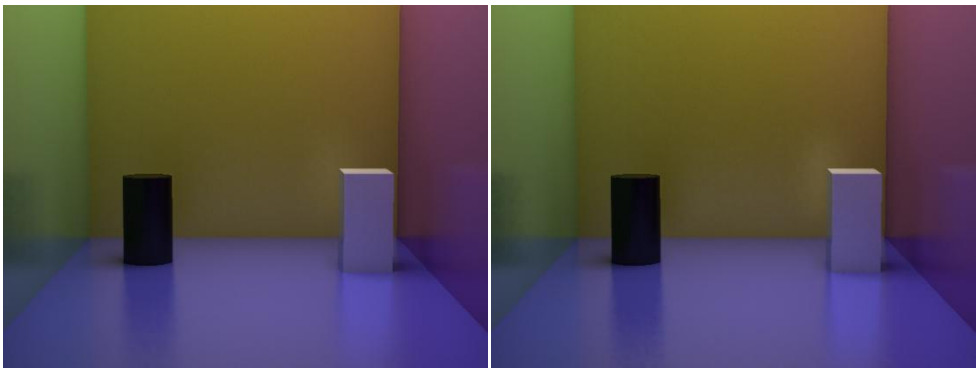

Figure 23: *Glossy reflections with 1 bounce using Blinn BRDF with 8 samples and shininess 300 using screen-space method with reflection cache created with shininess 1000 (left image) and screen-space method with correct shininess (right image).*

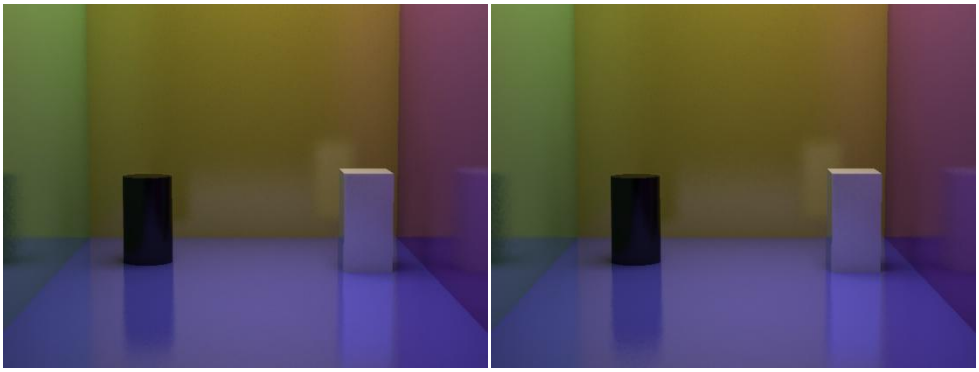

Figure 24: *Glossy reflections with 1 bounce using Blinn BRDF with 8 samples and shininess 3000 using screen-space method with reflection cache created with shininess 1000 (left image) and screen-space method with correct shininess (right image).*

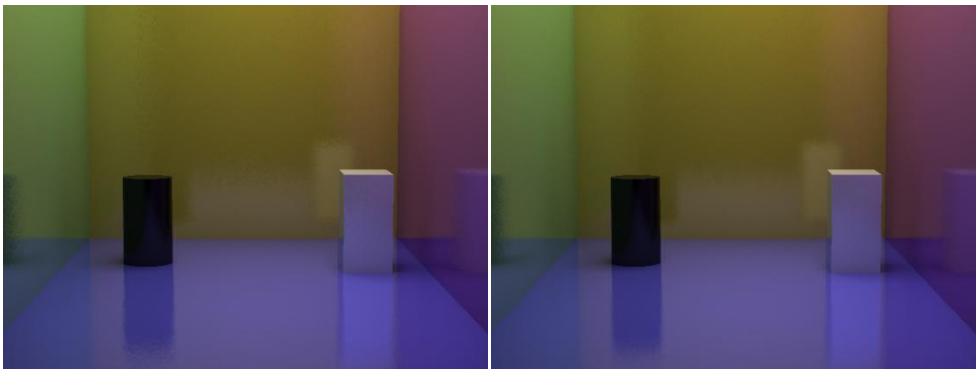

Figure 25: *Glossy reflections with 1 bounce using Blinn BRDF with 8 samples and shininess 5000 using screen-space method with reflection cache created with shininess 1000 (left image) and screen-space method with correct shininess (right image).*

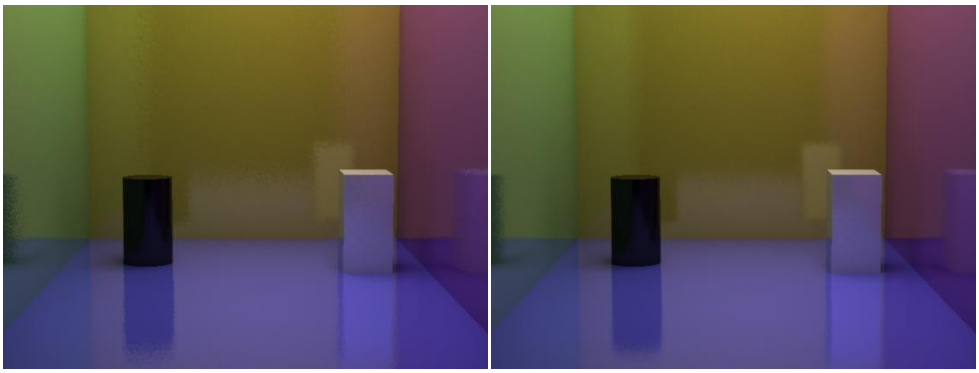

Figure 26: *Glossy reflections with 1 bounce using Blinn BRDF with 8 samples and shininess 10000 using screen-space method with reflection cache created with shininess 1000 (left image) and screen-space method with correct shininess (right image).*

As can be seen in the first image the approximation fails when the difference in shininess becomes too great. This is because the samples picked during the construction of the reflection cache do not extend far enough outside of the

cosine lobe use to create the reflection cache to accurately be used for very low shininess. The opposite effect can be seen in the last image. Pixels in which none of the samples fall within the significant part of the cosine lobe are colored using only low-weight samples, leading to pixel artifacts. Using extreme difference in shininess results in even greater distortions as exemplified with the figures below.

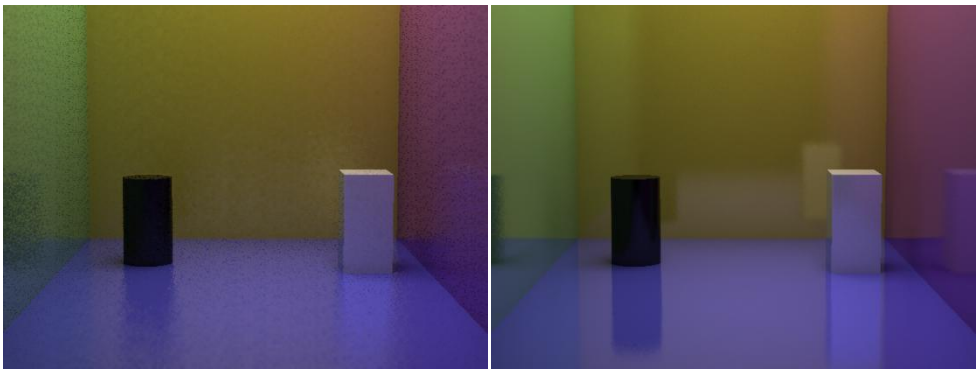

Figure 27: *Glossy reflections with 1 bounce using Blinn BRDF with 8 samples and shininess 10000 using screen-space method with reflection cache created with shininess 30 (left image) and screen-space method using correct shininess directly (right image).*

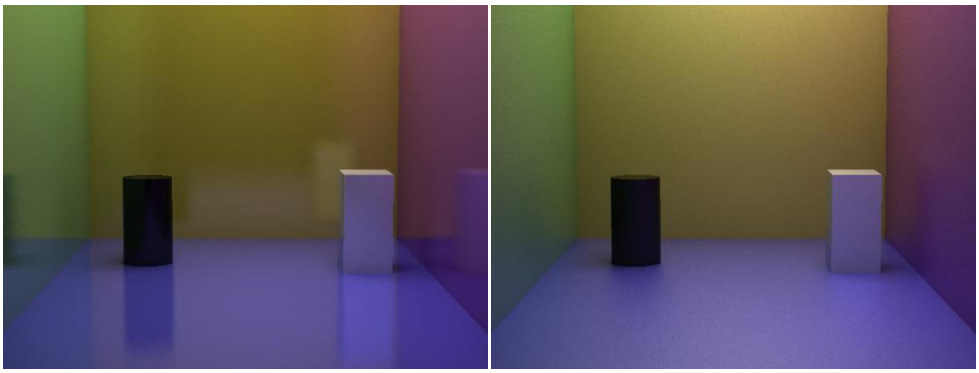

Figure 28: *Glossy reflections with 1 bounce using Blinn BRDF with 8 samples and shininess 30 using screen-space method with reflection cache created with shininess 10000 (left image) and screen-space method with correct shininess (right image).*

In the following example the image to the left is rendered by tracing the reflections using no bump mapping and rendering using the reflection cache with bump mapping by weighing the samples. The image to the right is rendered using V-Ray. The shininess remains unchanged at 300 between frames.

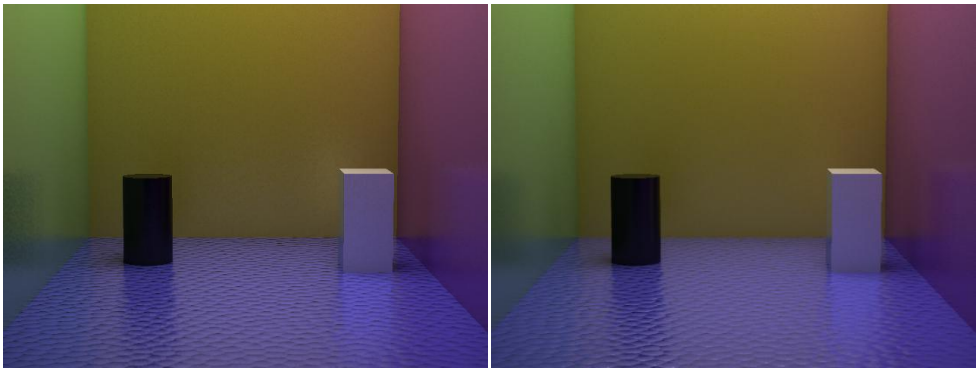

Figure 29: *Glossy reflections with 1 bounce and Bump Mapping using Blinn BRDF with 8 samples and shininess 300 using screen-space method with reflection cache created without Bump Mapping (left image) and V-Ray (right image).*

8.1.4 Real Life Example

Finally, below is an image rendered using the post-processor with buffers from the product configurator. The final image is shown together with the front color buffer.

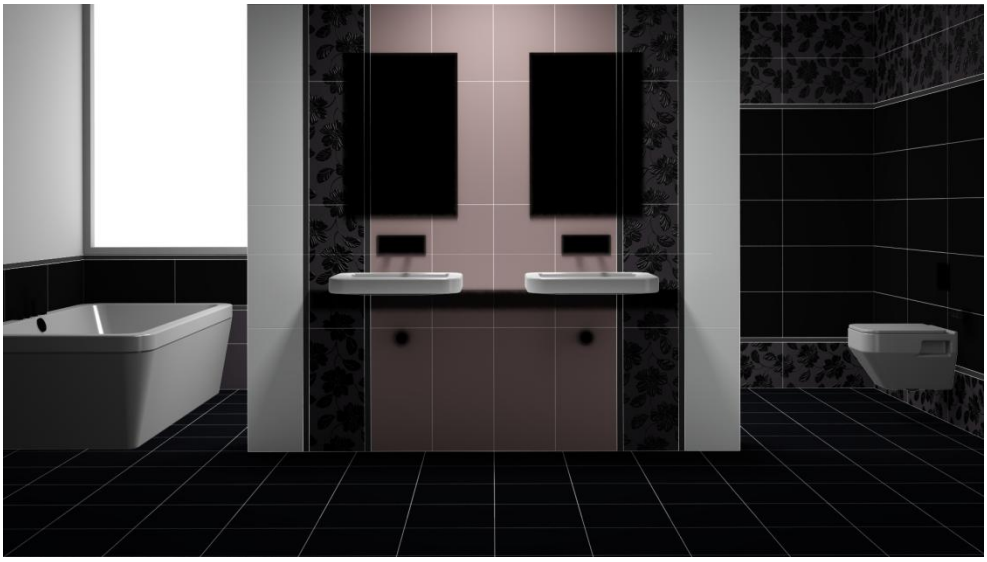

Figure 30: *Front Color Buffer C_f of a composed image of a product configurator.*

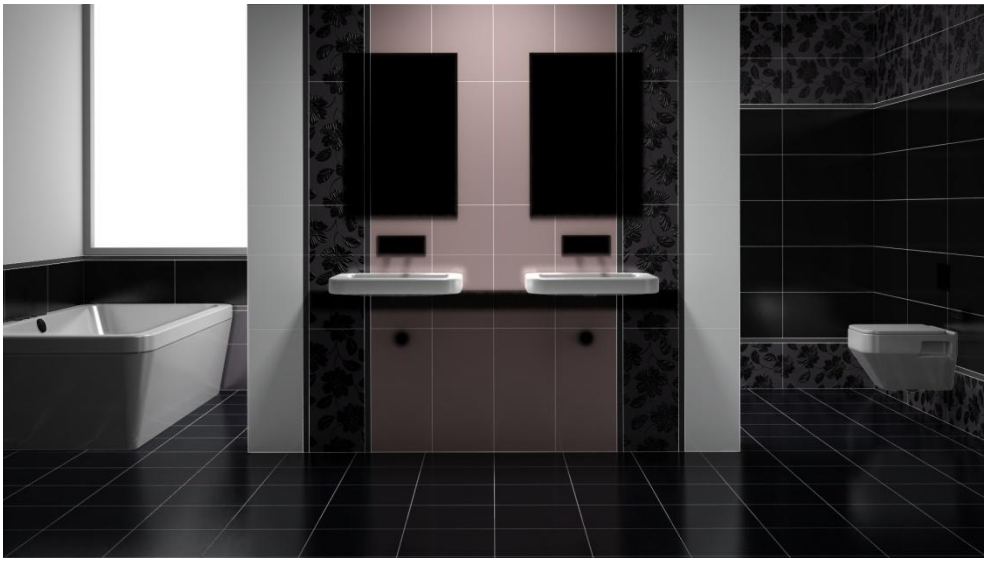

Figure 31: *Image with reflections of a composed image of a product configurator.*

8.2 Performance

The table below shows the performance of the post-processor.

Table 2. Performance based on renderings done using 8 GB of memory and two Intel Xeon E5450 3GHz CPUs with 640*480 buffers. Running times are averaged over 100 renderings.

Samples per pixel	Reflection Depth	Using Gloss	Using Blur	Using Different Shininess	Creating reflection cache (ms)	Rendering Reflection (ms)
1	1	No	No	No	2387	18
1	2	No	No	No	4323	23
1	3	No	No	No	5187	24
8	1	Yes	No	No	15964	53
8	1	Yes	Yes	No	15964	87
8	1	Yes	No	Yes	15964	83
8	1	Yes	Yes	Yes	15964	117

9 Discussion

This thesis presents an approximation scheme for ray traced reflections using screen-space data. Like any approximation the results are not entirely accurate. Objects that are neither part of the scene boundary nor visible from the camera cannot be captured in the reflections. Furthermore, the representation of the surrounding environment as a bounding box works well for the specific application but is difficult to apply to more general settings. Both of these issues stem from the same cause; the data available in screen-space is limited and reflected rays cannot always be traced. A more general method of handling screen-space misses would be an interesting problem. Several ideas came up during the thesis work. One approach would be to use screen-space sampling together with depth-preserving environment maps. The rays would then be sampled from the environment maps when screen-space sampling fails instead of sampling the scene boundaries. The basic idea is the same but does not put any restrictions on the type of scene being rendered while also capturing objects not visible from the camera. The usage of bounding boxes was preferred because it requires far less additional input and all the relevant scenes are bounded by boxes.

The texture completion algorithm used is tailored for the given application and any other application will undoubtedly need its own. Even for this application the texture completion algorithm fails severely for boundaries with heavily shadowed parts or varying structure. The technique was used in order to minimize the amount of user data needed. However, for quality reasons the boundary textures could instead be given as input buffers. This approach was attempted and rejected as lighting differs with the new viewing angle and difference between screen-space hits and misses becomes more obvious. The boundaries could be rendered from a different position but still have the lighting computed using the original camera position. Unfortunately, this is unavailable for the renderer used. Alternatively, any ray that hits the boundaries, even in screen-space, could instead be sampled from the pre-rendered textures. The reflection caching could be trivially extended to separate textures by also storing which texture to sample together with the coordinates. In addition to requiring more user-generated input data these pre-rendered bounding textures would also need to be composed of different component images, thus additionally increasing the running time.

The strength of reflections is not computed during the post-processing but handled as a predetermined value. Thus, the same factors are used for every bounce which ignores the dependence on incoming angle. The justification for this inaccuracy is that as the number of bounces increases the contribution is

consequently reduced and it is thus most important to accurately capture the strength of the first bounce. For this particular application this is further complicated by the fact that different techniques are used for different surfaces and sometimes several techniques are used in conjunction such as Fresnel reflections together with arbitrary fall-off textures or curves. As can be seen in the section *Results* this inaccuracy has minimal impact on the final color, even when the number of bounces increases. Regardless, this approximation does mean that the strength of additional bounces is arbitrary with respect to the physical justification for the numbers. A more accurate method would be to supply the surface properties used to determine the reflection strength as buffer input instead and calculate the strength, for each bounce and with respect to the new incoming angle, in real time.

Because the algorithm uses screen-space colors, any difference in lighting due varying incoming angles of the reflected rays, even for the first bounce, cannot be captured. This can be seen the section *Results* where the reflected color of the back side of the box to the right is significantly brighter compared to the images rendered by V-Ray. A more physically accurate and consistent method would be to have the color buffer contain color without any contribution from angle-dependent lighting and instead compute this lighting in the post-processor. This approach was not attempted as even when ignoring these effects the application is close to, if not at, the acceptable limit of running time. Furthermore, in order to improve image quality as many effects as possible should be rendered using V-Ray or some other commercial ray tracer.

An even more general approach to applying this technique would be to include it in a real-time graphics engine. The most obvious approach would be to create an engine using deferred shading as such engines already supply all of the buffers needed. Assuming indoor scenes the bounding quads could be acquired by rendering each of them using six pre-processing render calls. The geometry stage of such calls would likely be very fast as only two triangles would need to be rendered. These pre-processing steps could then supply the same buffers as the camera pre-processing step and the lighting of additional bounces could be handled identically to lighting of any other surface. With the use of a GPU and tolerance for worse visual quality the technique could potentially be used for games and other real-time applications.

The approximation scheme itself is capable of capturing local reflections of indoor scenes with changing geometry, a moving camera and dynamic surface properties but the reflection caching technique outlined puts severe limitations on frame-to-frame coherency. Unchanging geometry and camera settings by themselves

eliminate many possible applications. Furthermore, while reflection caching can handle arbitrary changes in reflection strength and color it performs best when bump mapping and shininess remain constant. The approximation scheme presented for handling changes of these values works well for small variations. However, the approximation fails when the variations are too great. If the hemisphere was sampled using uniform sampling rather than importance sampling any variation of both bump mapping and shininess could be captured. This would require significantly more samples however which is why it is not used.

10 Conclusion

Accurate rendering of local reflections in real-time is hard which is why most real-time graphics engines use different approximation techniques or ignore reflections altogether. Reflections can however be accurately captured using the off-line rendering technique of ray tracing and distributed ray tracing for glossy reflections. The method outlined is an attempt to approximate these techniques by screen-space sampling, thus making the running time independent of the complexity of the scene. To test for collision, a position is sampled at regular intervals along each reflection ray and projected to screen-space. The values of the position buffers at the projected position are then compared to the sampled position. Misses in screen-space are handled by using to classic ray tracing with a simplified scene, in this thesis work represented by a box with faces determined from the original screen-space buffers.

The approximation fails to capture object that are neither part of the pre-determined scene boundary nor visible from the camera and thus not present in the screen-space buffers. Furthermore, any lighting effects dependent on the incoming viewing angle are ignored. Reflection strength is given as input buffers and any reflection strength dependent on incoming angle will be incorrect for secondary reflection bounces. However, the method does not depend on this limitation and as such other applications could instead supply the post-processor with buffers containing the parameters needed to calculate the strength in real time.

Even using screen-space approximation the running time used to trace reflections in real time, especially for glossy reflections, was deemed too high. To circumvent this, a method for caching the reflection rays given static geometry is outlined. This caching can accurately be reused for any arbitrary change in lighting, diffuse color and reflection strength. The approximation used for changes in normal mapping and shininess however fails when these vary greatly between frames. No theorizing about the mathematical limits is attempted but a caching of shininess 1000 can give decent approximations of shininess as low as 300 and as high as 5000.

Given the limitations the approximation accurately renders both perfect reflections and glossy reflections with good visual quality. The reflection mapping technique allows the post-processor to be run at below 200ms which is fast enough for the intended application.

Based on the *Problem Statement* the following conclusions can thus be reached.

- Identify the auxiliary data needed

The projection parameters used to render the scene are needed to use the technique at all. To use boundary textures, either determined from screen-space data or given as input buffers, the view-matrix parameters are also needed if these coordinates are given in world-space.

At the very minimum color, positions and normal vectors of the front-side of objects are needed. If the strength of reflections is not constant over the scene a reflection filter used for this thesis work or the parameters needed to determine the strength are required as well. The color, position and normal vectors of the back side of objects are needed to accurately achieve higher reflection depth and correctly determining back side hits and reflection color of back side hits. In order to use different shininess and BRDFs in the scene buffers for these values are needed as well.

- Find a way to approximate the ray collision of ray tracing with screen space sampling

Only one approach is outlined but produces accurate results given the inability to capture reflections of objects not visible from the camera.

- Handling misses in screen space

The representation of the scene boundary as a box achieves a decent approximation for the relevant scenes but is difficult to generalize. The texture completion algorithm used fails even for some of the relevant surfaces.

- Handling glossy effects

Importance sampling can be used with good results similar to those of V-Ray for optimal scenes.

- Identify and remove or alleviate artifacts

Sampling artifacts can be alleviated using adaptive blur and colors that spill over can be handled by rendering the scene without anti-aliasing.

- Find applicable speed-up techniques

Reflection caching achieves running times below the interpreted maximum running time but comes with severe restrictions on frame-to-frame coherency.

11 Appendix A: Buffers

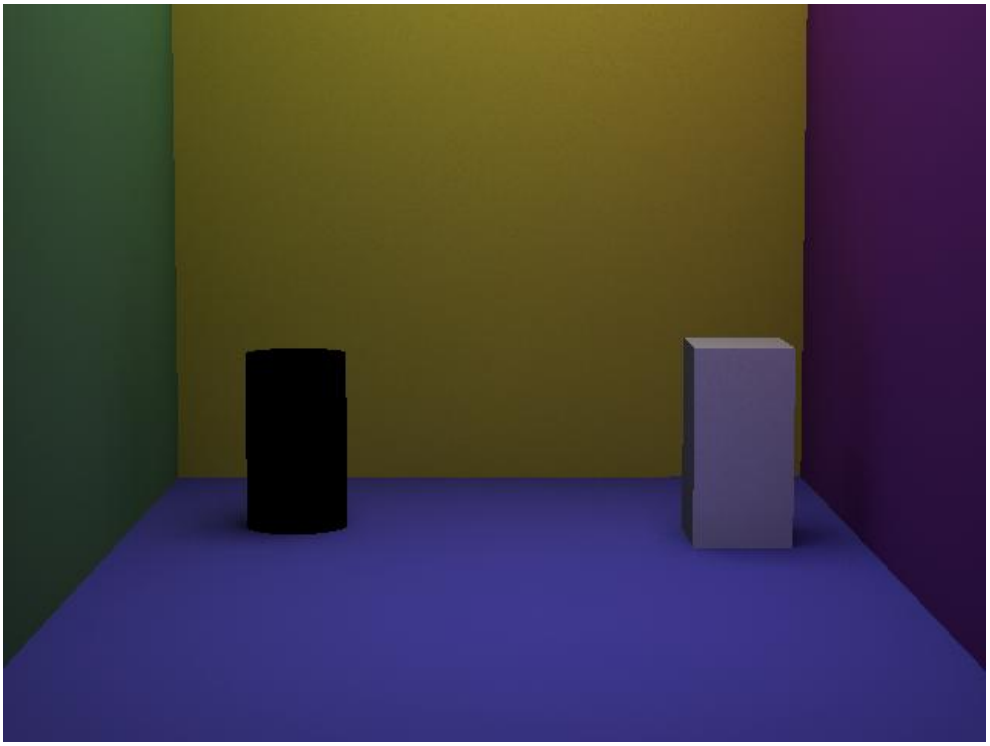

Figure A1: *Front Color Buffer* C_f

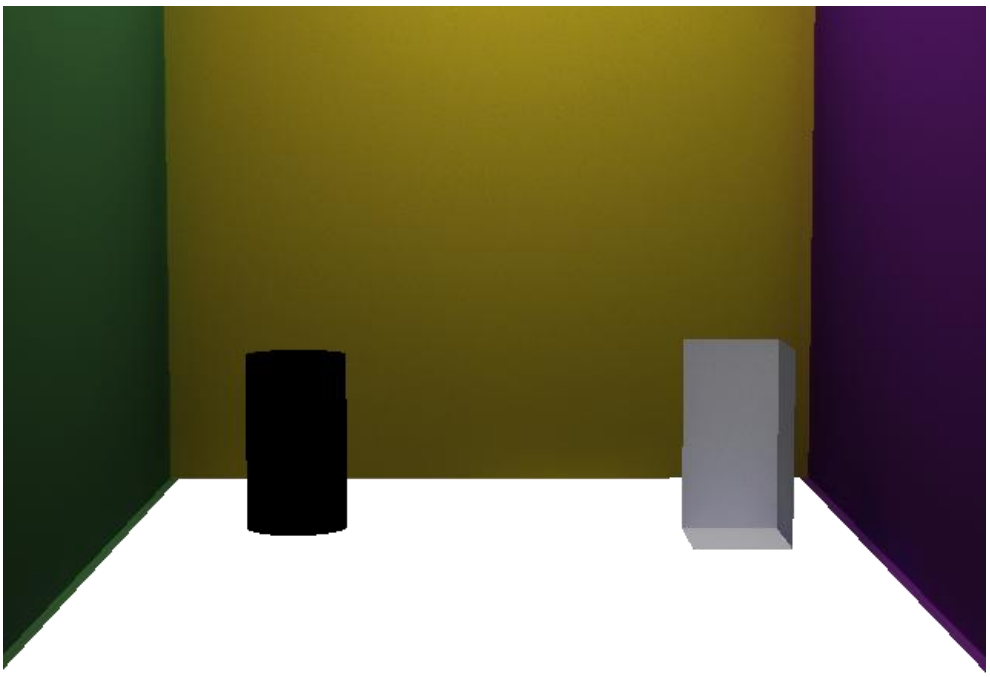

Figure A2: *Back Color Buffer* C_b

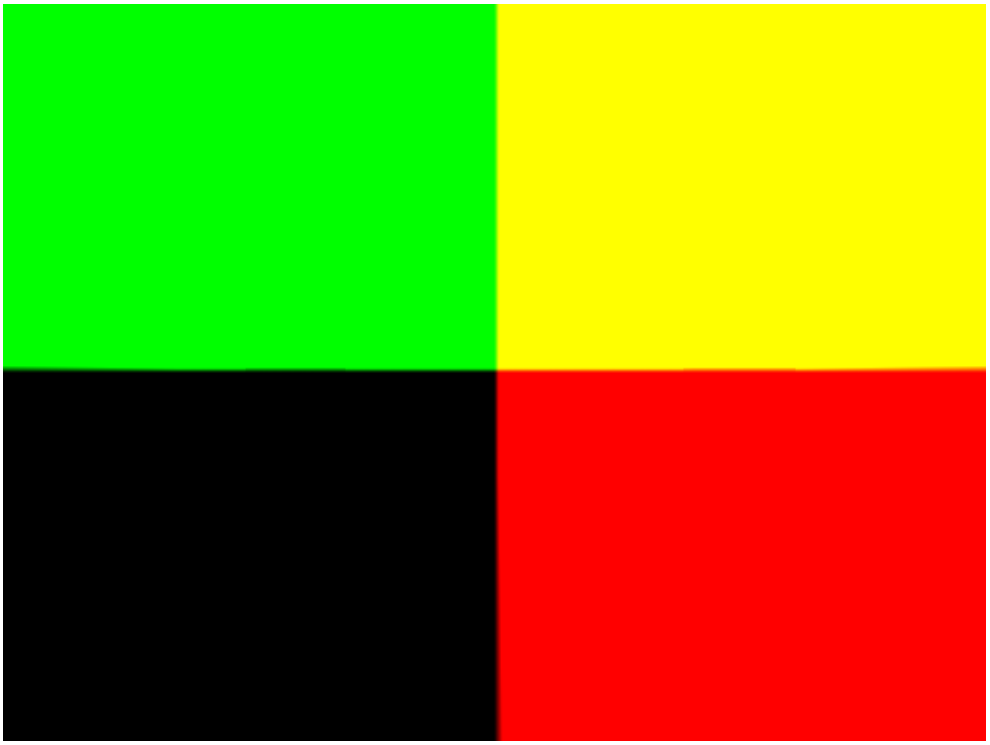

Figure A3: *Front Position Buffer* P_f

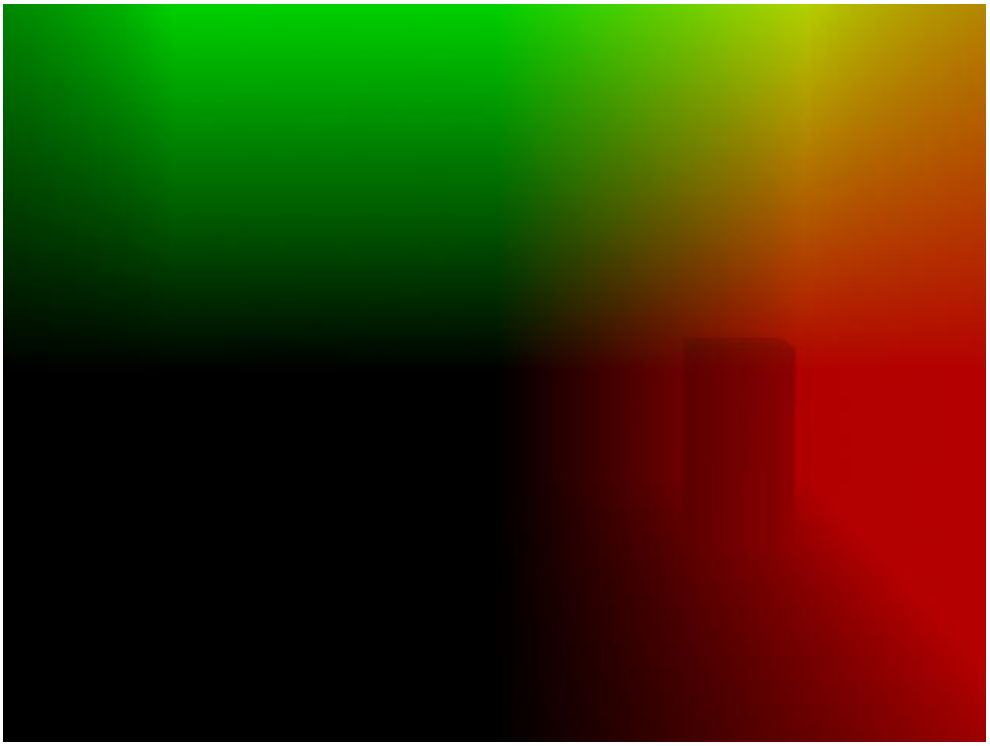

Figure A4: *Front Position Buffer factored with 0.01 (not used)*

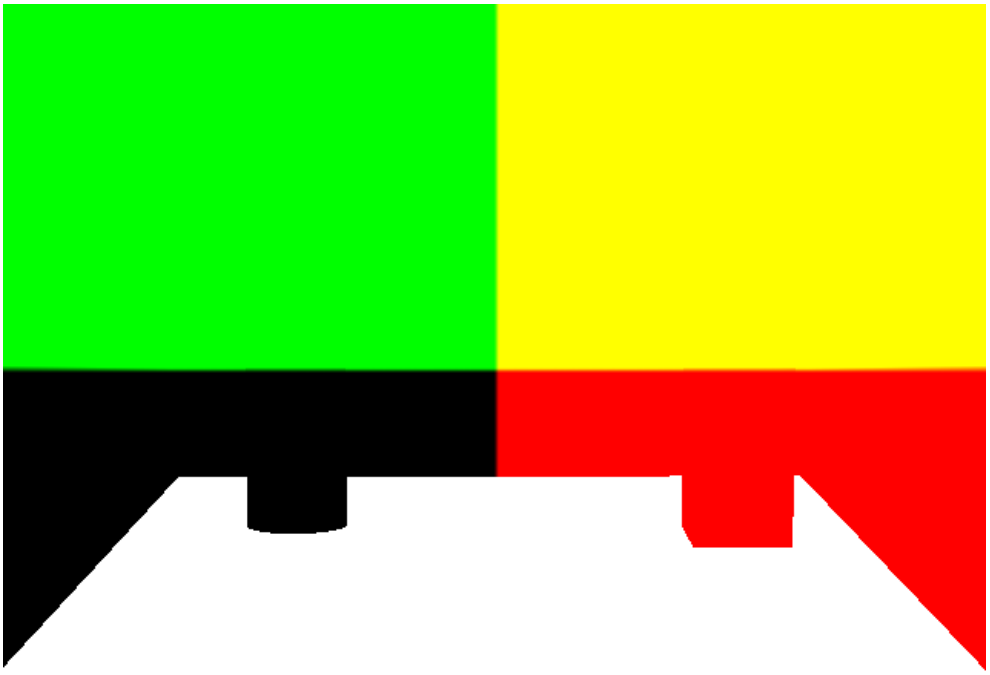

Figure A5: *Back Position Buffer* P_b

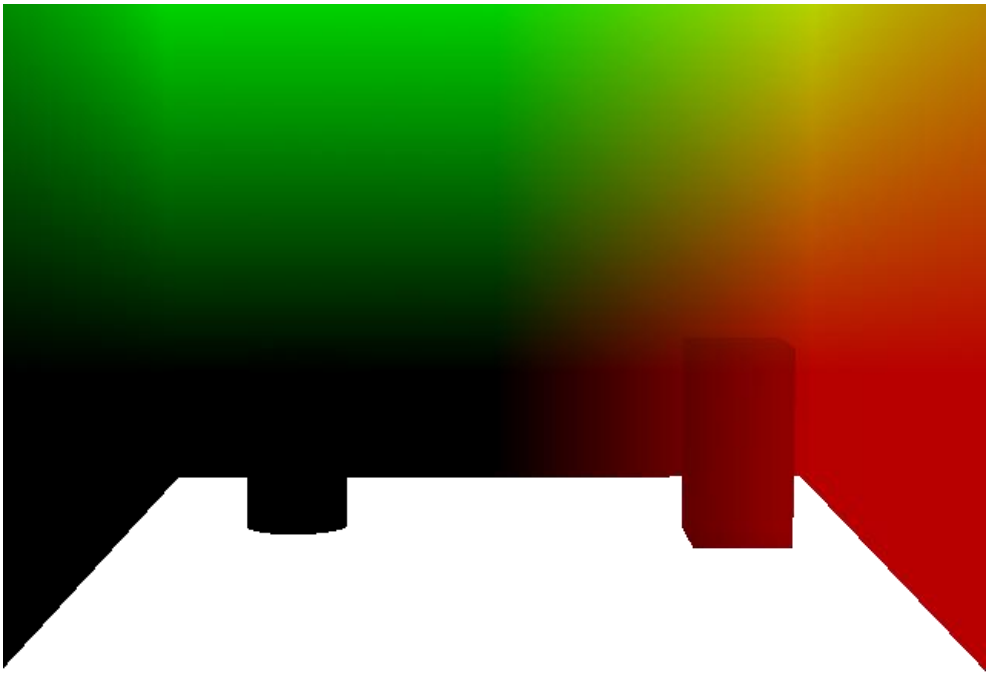

Figure A6: *Back Position Buffer factored with 0.01 (not used)*

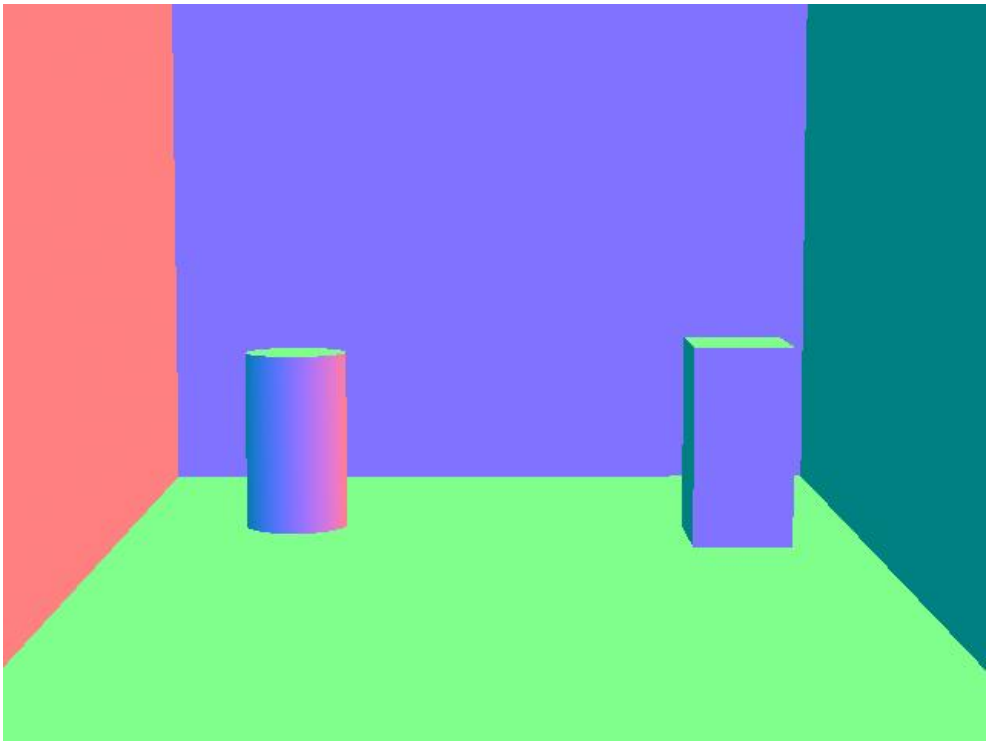

Figure A7: *Front Normal Buffer* N_f

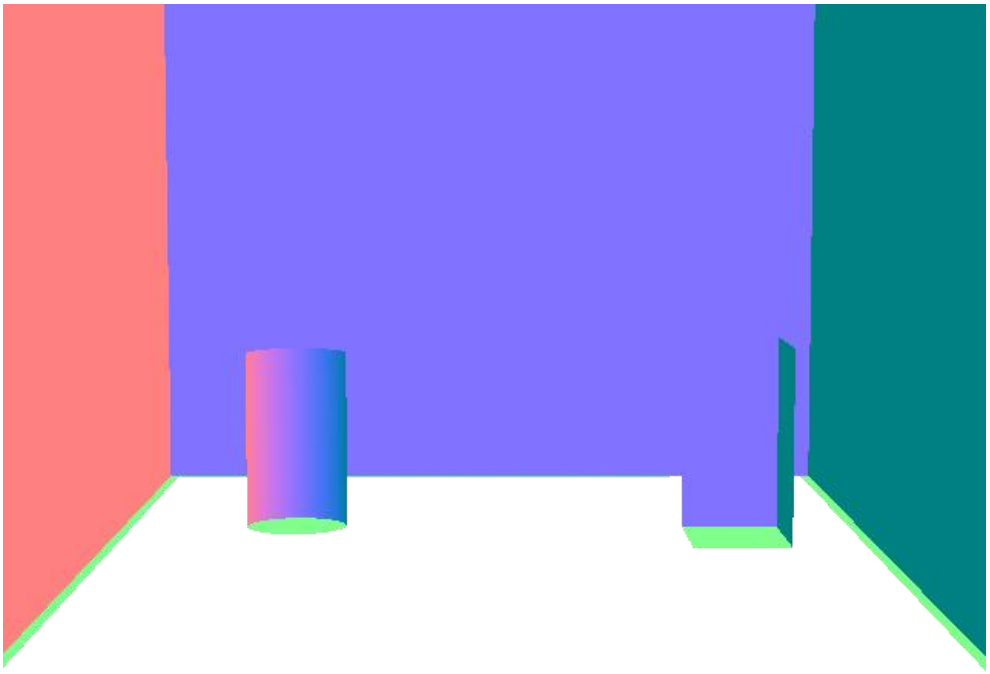

Figure A8: *Back Normal Buffer* N_b *(Note that the normals are inverted in order to use the renderer's back face culling to retrieve the back face data)*

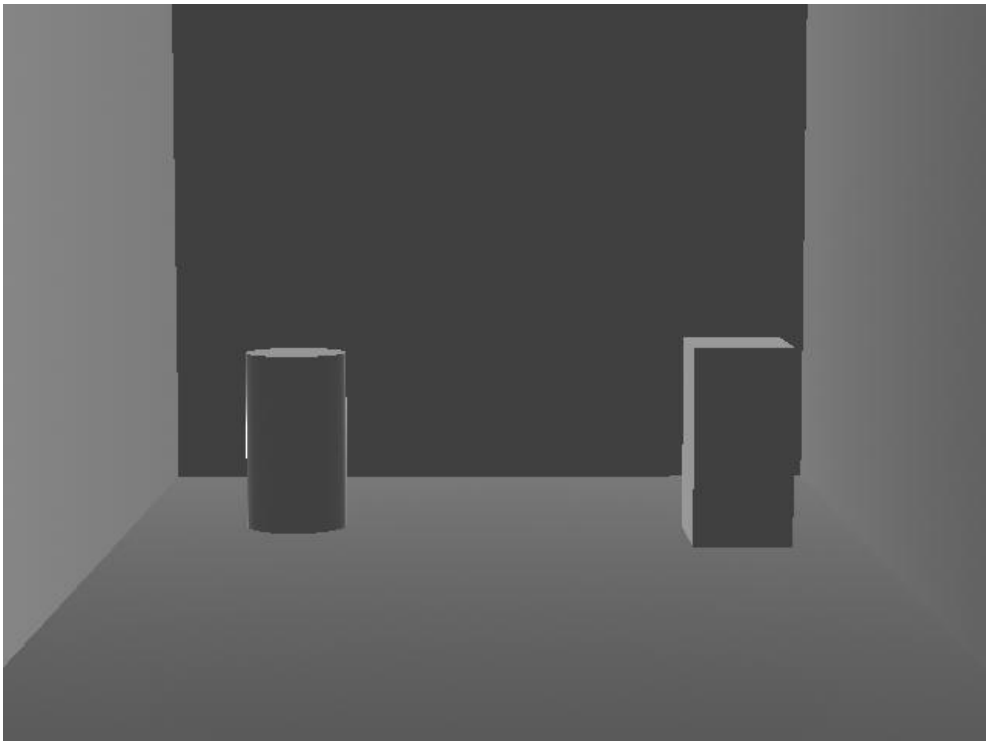

Figure A9: *Front Reflection Filter Buffer* F_f

Figure A10: *Back Reflection Filter Buffer* F_b

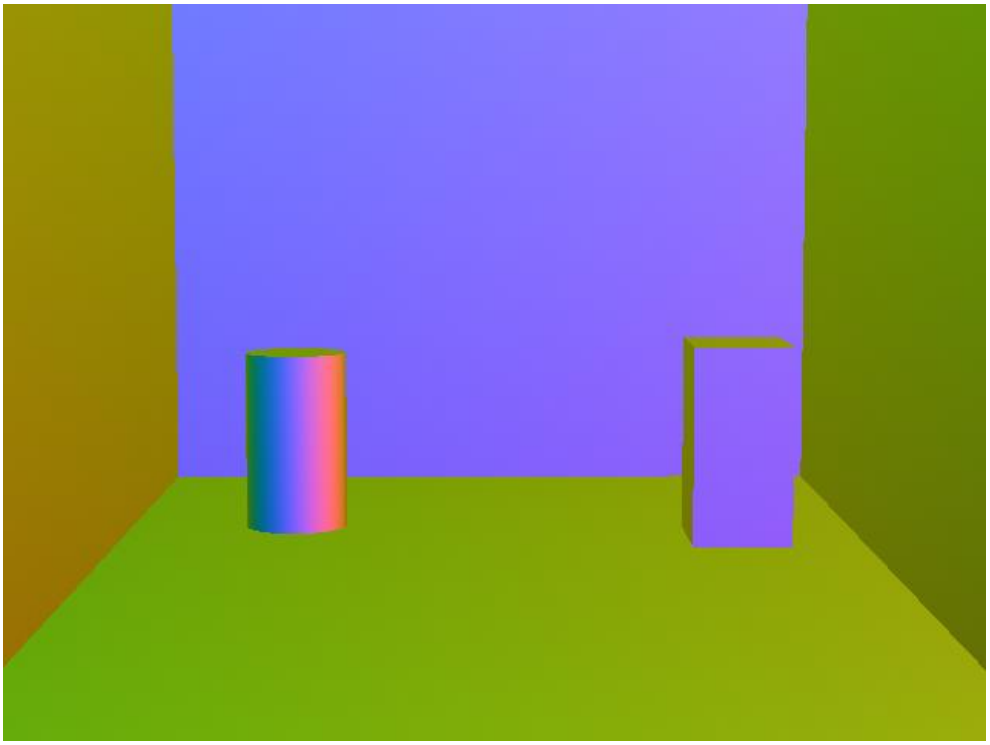

Figure A11: *Reflection Vector Buffer* R

12 Appendix B: Perspective Projection Matrices

The projection matrix used here is based on the OpenGL perspective projection matrix:

$$M_{P_{OpenGL}} = \begin{pmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{zFar + zNear}{zFar - zNear} & \frac{2 * zFar * zNear}{zNear - zFar} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Where

$$f = \text{cotangent}\frac{fovy}{2}$$

This is modified for three reasons. First, the field of view of the base renderer is given horizontally rather than vertically. Secondly, the base renderer attempts to mimic a physical camera. Two commonly used settings are vertical and horizontal shift, these are introduces to the matrix. Lastly, the $zFar$ and $zNear$ parameters are relevant for determining the size of the view frustum and to determine the depth value of a pixel, neither is needed for this application nor are the parameters themselves meaningful and as such these values are ignored.

$$M_P = \begin{pmatrix} f * \sqrt{h_s^2 + 1} & 0 & 0 & 0 \\ 0 & f * aspect\sqrt{v_s^2 + 1} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -h_s & -v_s & -1 & 0 \end{pmatrix}$$

Where

$$f = cotangent\frac{fovx}{2}$$

13 References

[1] Paul Joseph Diefenbach. Pipeline Rendering: Interaction and Realism Through Hardware-Based Multi-Pass Rendering. University of Pennsylvania, Department of Computer Science, Ph.D. dissertation, 1996.

[2] Kasper Høy Nielsen and NielsJørgen Christensen. Real-time recursive specular reflections on planar and curved surfaces using graphics hardware. University of We0073t Bohemia, Journal of WSCG, volume 10, pp. 91–98, 2002.

[3] Rui Bastos and Wolfgang Stürzlinger. Forward Mapped Planar Mirror Reflections. University of North Carolina at Chapel Hill, Computer Science Technical Report TR98-026, 1998.

[4] James F. Blinn and Martin E. Newell. Texture and Reflection in Computer Generated Images. Communication of the ACM. Vol. 19 no. 10, pp. 542-547, 1976.

[5] Ned Greene. Environment Mapping and Other Applications of W orld Projections, IEEE Computer Graphics and Applications, pp. 21-29, 1986.

[6] Brian Cabral, Marc Olano, and Philip Nemec. Reflection space image based rendering. SIGGRAPH ’99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques, pp. 165–170. ACM Press, 1999.

[7] Ziyad S. Hakura, John M. Snyder, and Jerome E. Lengyel. Parameterized environment maps. SI3D ’01: Proceedings of the 2001 symposium on Interactive 3D graphics, pp. 203–208. ACM Press, 2001.

[8] Jingyi Yu, Jason Yang and Leonard McMillan. Real-time reflection mapping with parallax. Proceedings of the 2005 symposium on Interactive 3D graphics and games, pp. 133-138, 2005.

[9] Jan Kauts and Michael D. McCool. Approximation of Glossy Reflection with Prefiltered Environment Maps. Graphics Interface 2000, pp. 119-126, 2000.

[10] Mark Colbertand and Jaroslav Křivánek. GPU-Based Importance Sampling. GPU Gems 3 Chapter 20, pp 459-475, 2007.

[11] Turner Whitteed. An improved illumination model for shaded display. CACM 23, 6, pp. 343–349, 1980.

[12] Henrik Wann Jensen et al. Monte Carlo Ray Tracing. Siggraph 2003 Course 44, 2003.

[13] Kevin Suffern. Ray Tracing from the Ground Up, Chapter 25, pp. 529-542, 2007.

[14] Matt Pharr and Greg Humphreys. Physically Based Rendering, from theory to implementation. Chapter 15.5.1, pp. 681-684, 2004.

[15] Vadim Konushin and Vladimir Vezhnevets. Automatic building texture completion. GraphiCon, pp. 174-177, 2007.

[16] Tomas Möller and Ben Trumbore. Fast, Minimum Storage Ray/Triangle Intersection. ACM Siggraph 2005 Courses, 2005.