Fachbereich 4: Informatik

Rendering view dependent reflections using the graphics card

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von

Guido Schmidt
guidoschmidt@uni-koblenz.de

Erstgutachter: Prof. Dr.-Ing. Stefan Müller
(Institut für Computervisualistik, AG Computergraphik)

Zweitgutachter: MSc. Gerrit Lochmann

Koblenz, im April 2013

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

. . . .

(Ort, Datum) (Unterschrift)

Zusammenfassung

Die Entwicklung der echtzeitfähigen Computergrafik ermöglicht mittlerweile immer realistischere Bilder und die Hardware kann dafür optimal ausgenutzt werden. Dadurch immer glaubwürdigere Lichtverhältnisse simuliert werden können. Eine große Anzahl von Algorithmen, effizient implementiert auf der Grafikkarte (GPU, auch *Grafikprozessor*), sind fähig komplexe Lichtsituationen zu simulieren. Effekte wie Schatten, Lichtbrechung und Lichtreflexion können mittlerweile glaubwürdig erzeugt werden. Besonders durch Reflexionen wird der Realismus der Darstellung erhöht, da sie glänzende Materialien, wie z.B. gebürstete Metalle, nasse Oberflächen, insbesondere Pfützen oder polierte Böden, natürlich erscheinen lassen. Dabei geben sie einen Eindruck der Materialeigenschaften, wie Rauheit oder Reflexionsgrad. Außerdem können Reflexionen vom Blickpunkt abhängen: Eine verregnete Straße zum Beispiel würde Licht, abhängig von der Entfernung des Betrachters reflektieren und verwaschene Lichtreflexe erzeugen. Je weiter der Betrachter von der Lichtquelle entfernt ist, desto gestreckter erscheinen diese.
Ziel dieser Bachelorarbeit ist, eine Übersicht über existierende Render-Techniken für Reflexionen zu geben, um den aktuellen Stand der Technik abzubilden. Reflexion entsteht durch den Einfall von Licht auf Oberflächen, die dieses in eine andere Richtung zurückwerfen. Um dieses Phänomen zu verstehen, wird eine Auffassung von Licht benötigt. Kapitel 2.1 beschreibt daher ein physikalisches Modell von Licht, gefolgt von Kapitel 2.2, das anhand von Beispielen ästhetisch wirkender Reflexionseffekte aus der realen Welt und den Medien die Motivation dieser Arbeit darlegt. In Kapitel 3 soll die generelle Vorgehensweise beim Rendern von Reflexionen deutlich gemacht werden. Danach wird in Kapitel 4 eine grobe Übersicht über existierende Ansätze gegeben. In Abschnitt 5 werden dann drei wesentliche Algorithmen vorgestellt, die zur Zeit oft in Spiel- und Grafikengines verwendet werden: Screen Space Reflections (SSR), Parallax-corrected cube mapping (PCCM) und Billboard Reflections (BBR). Diese drei Ansätze wurden zusammen in einem Framework implementiert. Dieses wird in Kapitel 5 vorgestellt und erklärt, gefolgt von detaillierten Beschreibungen der drei Techniken. Nachdem ihre Funktionsweise erklärt wurde, werden die Ansätze analysiert und auf ihre visuelle Qualität sowie ihre Echtzeitfähigkeit getestet.
Abschließend werden die einzelnen Verfahren miteinander verglichen, um ihre Vor- und Nachteile zu untersuchen. Außerdem werden die gewonnenen Erfahrungen beschrieben und Verbesserungsansätze vorgeschlagen. Danach wird ein kurzer Ausblick zur voraussichtlichen Entwicklung von Render-Techniken spekularer Effekte gegeben.

Abstract

Real-time graphics applications are tending to get more realistic and approximate real world illumination gets more reasonable due to improvement of graphics hardware. Using a wide variation of algorithms and ideas, graphics processing units (GPU) can simulate complex lighting situations rendering computer generated imagery with complicated effects such as shadows, refraction and reflection of light. Particularly, reflections are an improvement of realism, because they make shiny materials, e.g. brushed metals, wet surfaces like puddles or polished floors, appear more realistic and reveal information of their properties such as roughness and reflectance. Moreover, reflections can get more complex depending on the view: a wet surface like a street during rain for example will reflect lights depending on the distance of the viewer, resulting in more streaky reflection, which will look more stretched, if the viewer is located farther away from the light source.

This bachelor thesis aims to give an overview of the state-of-the-art in terms of rendering reflections. Understanding light is a basic need to understand reflections and therefore a physical model of light and its reflection will be covered in section 2, followed by the motivational section 2.2, that will give visual appealing examples for reflections from the real world and the media. Coming to rendering techniques: At first, the main principle will be explained in section 3 followed by a short general view of a wide variety of approaches that try to generate correct reflections in section 4. This thesis describes the implementation of three major algorithms, that produce plausible local reflections. Therefore, the developed framework is described in section 5, then three major algorithms are covered, which are common methods in most current game and graphics engines: *Screen space reflections* (SSR), *parallax-corrected cube mapping* (PCCM) and *billboard reflections* (BBR). After describing their functional principle, they are analysed of their visual quality and the possibility of their real-time applicability. Finally they are compared to investigate the advantages and disadvantages over each other.

In conclusion, the gained experiences are described by summarizing advantages and disadvantages of each technique and giving suggestions for improvements. A short perspective will be given, trying to create a view of upcoming real-time rendering techniques for the creation of reflections as specular effects.

Contents

1 Introduction 1

2 Real World Reflections 2
2.1 Physical background of reflections . . . 2
2.2 Characteristics of reflections . . . 4

3 Rendering Reflections 11

4 Approaches 12
4.1 Geometry transformation . . . 12
4.2 Image based approaches . . . 12
4.3 Ray tracing geometry . . . 14
4.4 Ray tracing sampled geometry . . . 15
4.5 Hybrid techniques . . . 16

5 Implementation 18
5.1 Framework . . . 18
5.2 Parallax-corrected cube mapping . . . 24
5.2.1 Algorithm . . . 24
5.2.2 Results . . . 26
5.3 Billboard reflections . . . 28
5.3.1 Algorithm . . . 28
5.3.2 Results . . . 34
5.4 Screen space reflections . . . 36
5.4.1 Algorithm . . . 36
5.4.2 Results . . . 45

6 Conclusion 49

List of Figures 50

Listings 52

List of Algorithms 53

References 54

1 Introduction

Often computer graphics aim to generate realistic looking images with complex lighting, including shadows and specular effects, modelled on the behaviour of light from the physical world. An important aspect to reach that goal is the reflection of light, whereby the surface properties of an object's material may change this behaviour. The real world has a wide range of materials, such as water, metals or glass, that reflect light differently, resulting in various visual effects. However, the possibilities of the classical rasterization pipeline supplied by the *Open Graphics Library* (OpenGL) was limited for a long time and therefore rendering specular effects was a difficult purpose. Accurate reflections have been reserved for ray tracing only, an image-synthesis approach, not able to run at real-time frame rates. This changed, when a more adjustable pipeline, providing programmable shader units, was introduced with version 2.0 of *OpenGL*. It now reached version 4.3, whereby new features were integrated with every version. Since then, the hardware, not only graphics hardware, but also the CPU, evolved dramatically and still is. These two factors allowed real-time ray tracing approaches on the GPU as well as on the CPU on the one hand and more complex techniques to enrich the rasterization pipeline on the other hand. Thus, many ideas and techniques for rendering reflections arose over the years. Some only use the classical rasterization pipeline, some utilize ray tracing approaches, implemented on the graphics hardware. Hybrid techniques try to gain the best of both worlds, whereby a lot of advanced ideas and approaches were developed over the last few years, providing better results for real-time reflections. This thesis aims to give an overview of existing render techniques, that make use of the programmable graphics hardware and will describe three common algorithms, often used in graphics applications lately, in detail. For testing and comparison of the chosen techniques, a framework and its implementation, using the current *OpenGL* API, is described.

2 Real World Reflections

To first understand reflections and how they occur, understanding light, how it behaves and how it hits a surface, getting absorbed or reflected, is of primary importance. Therefore, section 2.1 introduces a short model for better understanding light followed by a motivational section featuring appealing examples of reflection effects.

2.1 Physical background of reflections

Without light, you could not see anything in real world and therefore light is the most essential part to understand reflections. By using the term *reflection*, usually *reflection of light* is meant, as there are other kinds of reflections, e.g. of acoustic waves or other physical radiations. Light is an electromagnetic radiation, called *visible light* at a range of ~380 nm to ~740 nm. To get a better understanding of light, the model of *photons* is commonly used: The amount of light can be seen as a large collection of photons, whereby every photon is a small quantum of light with a position and a direction, a speed (the speed of light, only depending on the medium, it goes trough) and obviously a wavelength, as it has to be in the above-mentioned spectrum. A photon also has an amount of energy q, which can be summed up to the total amount of energy Q of a collection of photons, called *spectral energy*, depending on the wavelength λ. Q has to be seen as a density function, that gives the energy-density at a given interval of wavelengths (For more detailed information, see [SAG$^+$05]) and can be expressed with:

$$Q = \frac{\Delta q}{\Delta \lambda} \tag{1}$$

Because the power of a light source may change over time and thus the amount of photons would change over time, the energy is measured during a time-interval Δt.

$$Q = \frac{\Delta q}{\Delta \lambda \Delta t} \tag{2}$$

In terms of reflections, another interesting question is, how much light hits a certain area on a surface. This question is answered by the quantity of *irradiance*. Looking at a single point in the world is not possible, because mathematically this would be an infinitesimal small area and no photon would hit it. We rather have to look at an area on a surface, to find out how much light arrives there. To reveal information about the amount, a surface receives, a finite area ΔA is used to measure the incoming light. So the amount of energy per unit area gets:

$$Q = \frac{\Delta q}{\Delta \lambda \Delta t \Delta A} \tag{3}$$

Although *irradiance* gives information about the amount of light, that reaches a surface, it doesn't provide information about the direction, the light comes from. To get the additional information about the direction, one has to use the quantity of *radiance*. We can think of a light measuring device, that has a conical formed filter applied, that lets only photons pass, that come from a certain direction. Thereby the consequential amount of energy is also dependent on the incoming angle $\Delta\sigma$:

$$Q = \frac{\Delta q}{\Delta \lambda \Delta t \Delta A \Delta \sigma} \tag{4}$$

Usually *Radiance* is the amount, computed in graphics applications. To characterize how materials reflect light, usually the term *bidirectional reflectance distribution function* (BRDF) is used. The BRDF is a function that describes how light is reflected by a certain material and is often approximated by analytical functions. Common BRDF models are for example the *Lambert* material, that models a perfect diffuse surface, or the *Phong* material, creating small highlights like they occur on plastic materials.

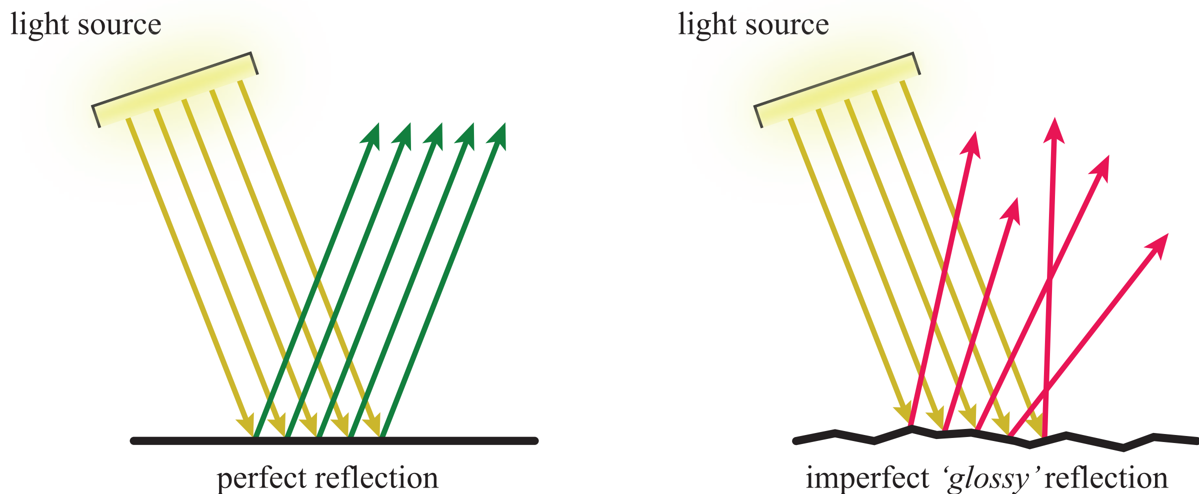


Figure. 1: Left: perfect reflection on a complete flat surface, **right:** imperfect reflection caused by the natural roughness of the surface material

A perfect specular reflection (*mirror reflection*) would result from a 100 percent even and reflective material, where light would be reflected without a loss of energy, equal for every wavelength, respectively every color. But in the real world, a certain amount of energy gets lost, if light hits a surface and gets reflected. This energy-amount changes at different wavelengths, e.g. a piece of gold reflects yellow more than blue and so the visual appearance is affected to look more orange-yellowish. Real surfaces are not completely even. They always have a certain roughness and thereby do not reflect the light perfectly, but rather scatter the reflected light in different directions as shown in figure 1. Such imperfect reflections are often called *glossy*. Depending on the kind of surface, *glossy reflections* look blurred or stretched, whereby the effects are distinguished as *isotropic* or *anisotropic* reflection. To get a basic understanding of what *isotropic* and *anisotropic* mean, take a look at their definitions:

> **isotropic:**
> *adjective*, Of equal physical properties along all axes. [1]
>
> **anisotropic**:
> *adjective*, Of unequal physical properties along different axes. [2]

In terms of reflections, isotropic would look stretched equally in every direction, so the reflected image would look more blurred and anisotropic reflections would look more stretched in only one certain direction, as compared in figure 2. This effect is caused by tiny random bumps in the material surface. This bumps may be possibly so small, that they can not be recognized with the human eye, but the effect affecting the reflected image caused by them will be. Brushed metals are an excellent example for materials with noticeable bumps, like the brushed metal pipe in figure 3 on the left. Summarized, the look of reflections highly depends on the surface material they occur on.

[1] `dictionary.reference.com/browse/isotropic`, [accessed 18/03/2013]
[2] `dictionary.reference.com/browse/anisotropic`, [accessed 18/03/2013]
[3] `www.neilblevins.com/cg_education/aniso_ref/aniso_ref.htm`, [accessed 03/04/2013]

Figure. 2: Blurred reflection in *Unreal Engine 3* tech demo *Samaritan*, **top:** different amounts of anisotropic blurred reflection in y-direction, **bottom:** different amounts of isotropic blurred reflection (images: [MD11])

Figure. 3: **Left:** close-up of brushed metal pipe railing, **middle:** anisotropic reflection on a fridge door, **right:** anisotropic reflection of traffic cones caused by multiple puddles on the street, (images: [3])

2.2 Characteristics of reflections

The look of reflections depends on a variety of properties. After section 2.1 described a physical model of light and how it reflects from surfaces, this section will give a visual appeal of characteristic looks of reflections from the real world, art and media.

The most noticeable effect should be visible in rainy nights, when light and shiny object are reflected in wet pavements and streets. The missing daylight is the reason, that reflections at night are more recognizable as at daytime. Puddles on the surface of the urban environment reflect street lanterns, traffic lights, luminous advertising and other city lights and thus creating a colored, high contrast look (see figures 4 and 5), that gives them and interesting appeal for the human eye. Even on daylight, puddles reflect their surrounding environment, often less noticeable as in night times (see figure 6), because the scattered sunlight, which lights the scene through the clouds, reduces the contrast of the reflections compared with the rest of the world.

[4] seblagarde.wordpress.com/2012/12/10/observe-rainy-world/

Figure. 4: High contrast reflections on wet pavement at night (images: [4])

Figure. 5: Wet street materials reflecting city lights at night (images: [5])

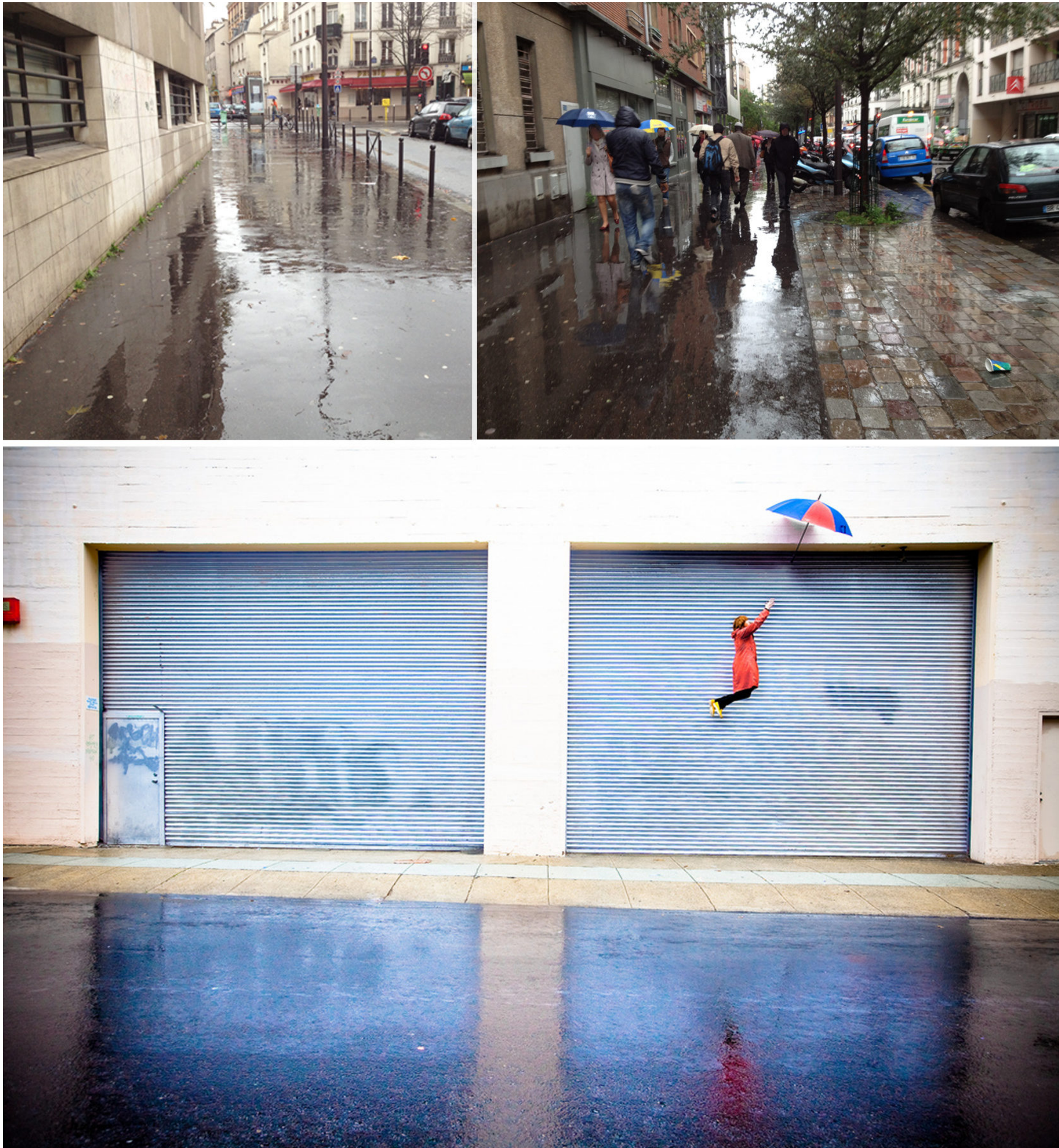


Figure. 6: Street reflections at daylight (images: [6])

In general, water or thin puddles always cause nice looking reflections. The surf on a beach for example moistens the first few meters of the shore, generating a thin layer of water on it, which reflects the objects around it nearly mirror-like (see figure 7, left side). Leftovers from the last rain showers on a sunny day will create similar effects on the pavement (see figure 7, right side). More deeper water will reflect even more mirror-like, depending on the light and weather situation: everyone has seen one of the impressing pictures of big city's bays or rivers, reflecting the city-lights in long streaky, blurred reflections (see figure 8, right).

[5]left: www.flickr.com/photos/26226522@N08/3517936516, [accessed 25/03/213],
top right: 2.bp.blogspot.com/-HD-9wy1PSvM/TaXIdMGCStI/AAAAAAAADpc/snOnrO21LTE/s1600/city2.jpg, [accessed 25/03/2013],
bottom right: vitofun.net/the-fourth-estate/solitude/8010894, [accessed 25/03/2013]
[6]top right and top left: seblagarde.wordpress.com/2012/12/10/observe-rainy-world/, [accessed 25/03/2013],
bottom: Nate Bolt, www.flickr.com/photos/boltron/4181707479/, [accessed 25/03/2013]
[7]seblagarde.wordpress.com/2012/12/10/observe-rainy-world/, [accessed 25/03/2013]

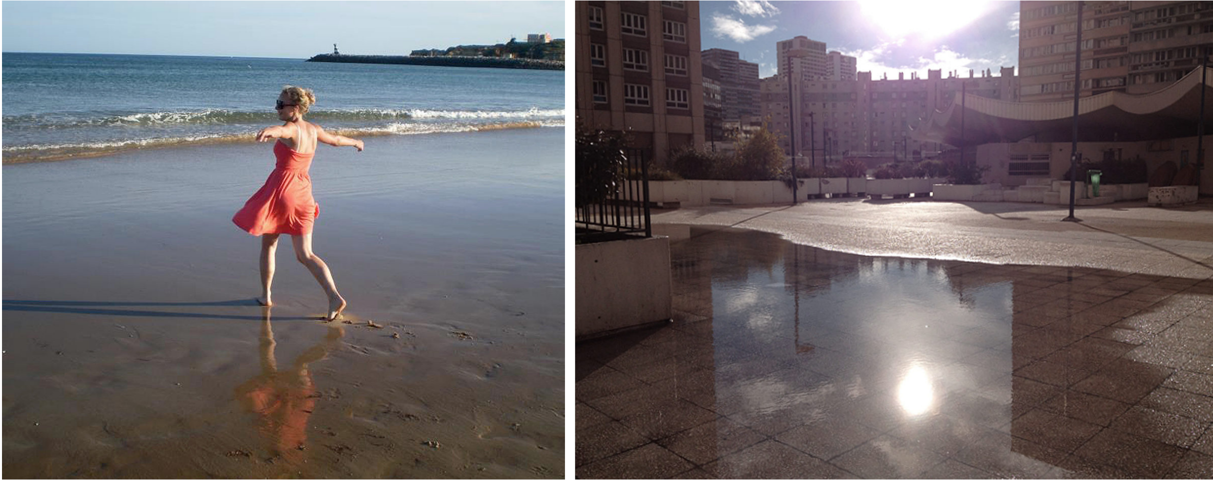


Figure. 7: Left: Reflection of a young woman in the thin water layer on the shore, **right:** puddle in an urban environment (left image: [7])

Figure. 8: Left: Reflections on a channel in Venice, **right:** Stockholm's bay reflects the city's lights at sunset (images: [8])

But not only wet surfaces cause reflections, there are many more materials, that reflect the light and give beautiful reflection images. Besides the discussed watery surfaces like lakes, the sea, puddles etc., metals can appear very glossy: e.g. brushed metals or car paint (see figure 9, left). Sometimes even floors reflect the world, especially when they are polished, as it is often in museums, hotels or public buildings (see figure 9, right).

Figure. 9: Left: A Maybach parked in New York city reflects the environment, **right:** Polished floor at the Photo Festival in Seoul (2011), *The Detour of the Real*, Seoul Art Museum (images: [9])

[8]left: `upload.wikimedia.org/wikipedia/commons/a/a0/Venice_-_Water_Reflections.jpg`, [accessed 25/03/2013],
right: `interfacelift.com/wallpaper/D47cd523/02683_stockholm_1920x1200.jpg`, [accessed 25/03/2013]

Regarding big cities, modern architecture tends to use more reflective materials, such as glass and metals, that create a high reflective modern city-environment as seen e.g. in figure 10.

Figure. 10: Left: *Cloud Gate* by artist *Anish Kapoor* on the AT&T Plaza in Chicago, **right:** Mirrored windows of a building reflects a set of modern skyscrapers (images: [10])

Reflections also seem to fascinate artists around the globe for a very long time. Impressionistic artists like Claude Monet or Vincent van Gogh used the image of reflecting surfaces in their paintings (see figure 11). Photographers took and take pictures, showing interesting subjects twice because of a reflected image somewhere. Or they even take the reflection-surface as their second lens and show the world from the view of a reflection as in the *London in puddles* series by Gavin Hammond (see figure 12, right). Even in modern art, reflection can create beautiful experiences, speaking of such as of Taro Shinoda's *Lunar Reflections* or of samples from the work of visual artist Rafaël Rozendaal (see figure 13). Last but not least, computer- and console-games are geared toward more realistic looking graphics, including real-time reflections, as for example seen in *Crysis 2* (see figure 14).

Figure. 11: Left: Vincent Van Gogh - Starry Night Over The Rhone (1888), **right:** Claude Monet - Argenteuil. Yachts (1875) (images: [11])

[9]left: `upload.wikimedia.org/wikipedia/commons/f/f1/Maybach_car_in_New-York_city.JPG`, [accessed 26/03/2013], right: Thomas Wrede, `thomas-wrede.de/images/455.jpg`

[10]left: `www.atlasls.com/images/atlas-chicago.jpg`, [accessed 24/03/2013], right: `paulapuffer.net/wp-content/uploads/2012/10/October-1-0706.jpg`

[11]left: `upload.wikimedia.org/wikipedia/commons/9/94/Starry_Night_Over_the_Rhone.jpg`, [accessed 26/03/2013], right: `allart.biz/photos/view/Oscar_Claude_Monet.html`, [accessed 22/03/2013]

Figure. 12: Left: *Behind the Gare Saint-Lazare* (1932) by photographer Henri Cartier Bresson, **right:** *London in puddles* (2012) by photographer Gavin Hammond, (images: [12])

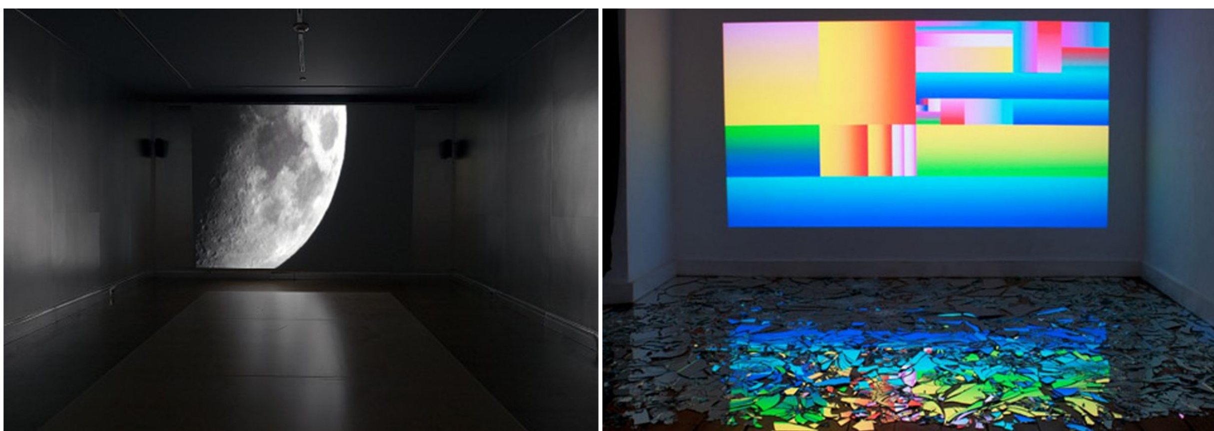


Figure. 13: Left: *Lunar Reflections* (2010) by conceptual artist Taro Shinoda, **right:** Sample from exhibition *Everything Dies* (2012) by visual artist Rafaël Rozendaal, (images: [13])

All in all, there are many reasons to bother about reflections in real-time graphics. The variety of reflecting materials from water, metals and glass to polished surfaces, like floors or tables gives a big range of different effects, that may be ambitious to achieve in a virtual environment. Since computer graphics tend to get more realistic, reflections are a big gain in realism.

[12]left: afterimagegallery.com/bressonbehind.htm, [accessed 26/03/2013],
right: gavinhammond.tumblr.com, [accessed 25/03/2013]

[13]left: aesthetic.gregcookland.com/2010_01_24_archive.html, [accessed 26/03/2013],
right: kunstverein-arnsberg.de/rafael-rozendaal-2, [accessed 26/03/2013]

[15]www.heise.de/newsticker/meldung/GDC-Unreal-Engine-4-zeigt-filmreife-Render-Szenen-in-Echtzeit-1832905.html, [accessed 30/03/2013]

Figure. 14: *Crysis 2* was one of the first computer-games, that implemented real-time local reflections (image: [14])

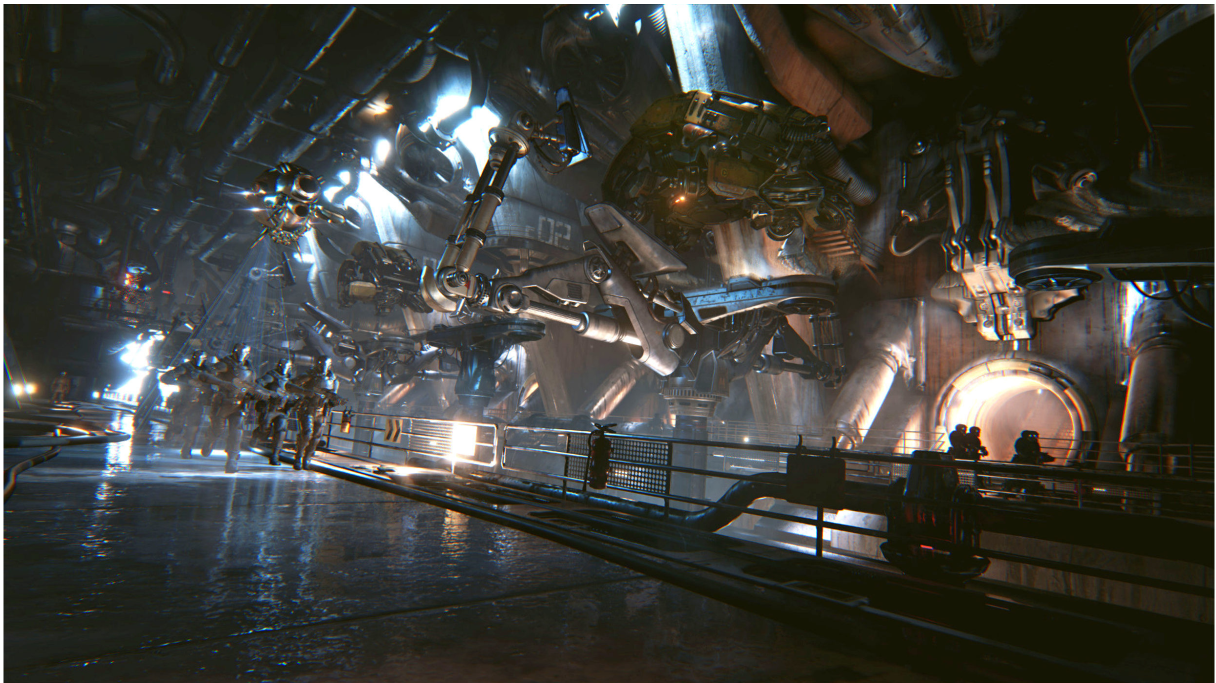


Figure. 15: Lately released tech demo *Infiltrator* showing *Unreal Engine 4*'s qualities, including real-time reflections (image: [15])

3 Rendering Reflections

Before going into any details, this section will give a general idea of the concept for rendering reflections. The following image illustrates the idea, which will describe the general approach for most render techniques, that generate reflections.

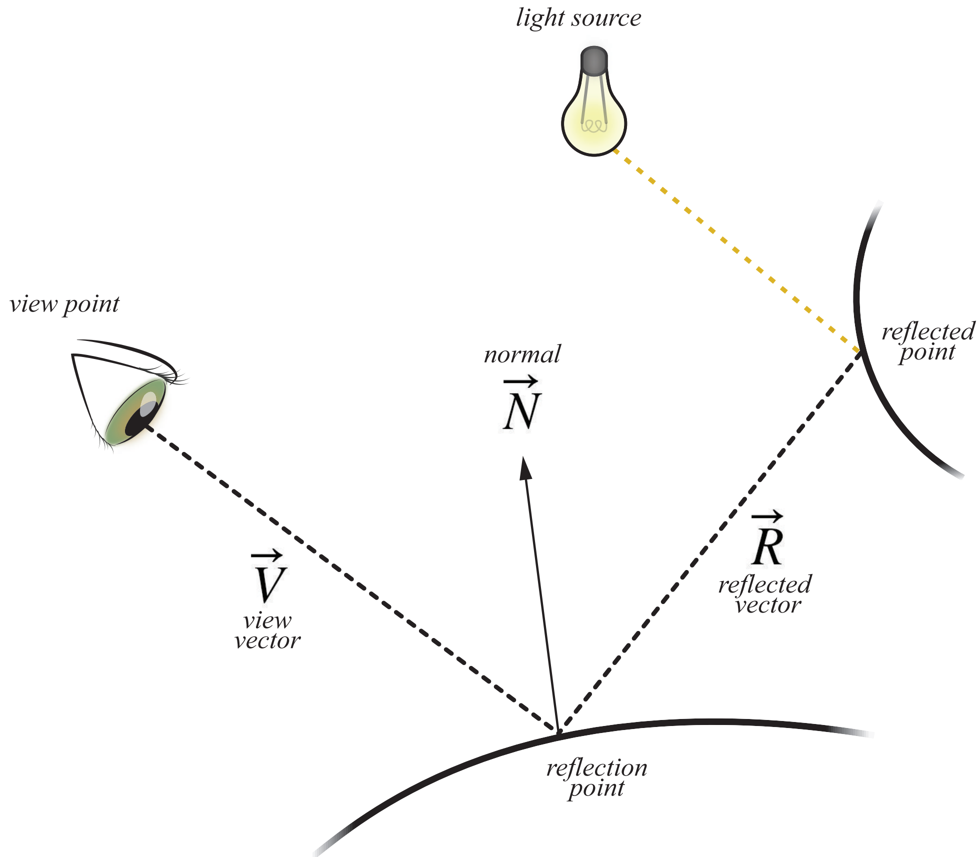


Figure. 16: Basic idea of tracing the view ray to find the reflected point

Thinking about the physical world, rendering reflections can be seen as the collection of light paths from a light source, that are reflected from a surface more than once and end up in the eye of the viewer, as described in 2.
From a more mathematical view, we can describe the process as looking along the *view vector* $\overrightarrow{V}$ to first find the *reflection point* as it is done in [SKUP+09] or [MT97]. Reflected at *normal* $\overrightarrow{N}$, the *view vector* results in the *reflected ray* $\overrightarrow{R}$, which is followed to find the *reflected point*, where the radiance, received from the *light source*, is reflected back at the *reflection point* towards the view position, respectively the camera (see figure 16). This process can be thought of tracing a ray from the *reflection point*, which will be the ray origin O, in direction $\overrightarrow{D}$ to get the illuminated color at the intersection of the ray and the *reflected point*. Expressed in a parametric equation, this results in:

$$\overrightarrow{R}(t) = O + t \times \overrightarrow{D} \tag{5}$$

The equation defines the reflected ray $\overrightarrow{R}(t)$, with O as the origin of the ray and $\overrightarrow{D}$ as the direction of the ray, in which it is reflected. By changing parameter t, every point along this reflected ray can be evaluated. The above shown image would result in an ideal specular reflection (or *mirror reflection*).

4 Approaches

Before describing the implemented approaches, this section will give a brief overview of the state-of-the-art techniques for rendering reflections. The most established methods will be shortly explained, since all three implementations, described in section 5, are based on older ideas or common methods for reflection-generation in computer graphics today. To better separate all the different approaches for rendering reflections, [SKUP+09] classifies algorithms in four major categories, based on their main ideas:

1. **Geometry transformation**
   The scene geometry is transformed into the corresponding mirrored vertices, that are seen in the reflector.

2. **Image based approaches**
   A part of the scene is represented using images. The reflected ray is then used to sample these images and get the reflected color, e.g. the classical environment mapping method is image based.

3. **Ray tracing geometry**
   The scene is ray traced directly. Traditional world space raytracing algorithms can be adapted.

4. **Ray tracing sampled geometry**
   The geometry of the scene is sampled and stored in textures, which are used to compute the intersection with the ray.

4.1 Geometry transformation

One of the first ideas to generate local reflections was to mirror the scene geometry at the reflector and render it again using the stencil buffer to mask out hidden objects. This is trivial and works well for planar reflections, but becomes a more complex problem, when using non-planar reflectors. There are techniques to create reflections on curved objects using geometry transformation, e.g. the technique described in [OR98]. Using geometry transformation is an old approach, therefore not often used anymore and can be seen as outdated.

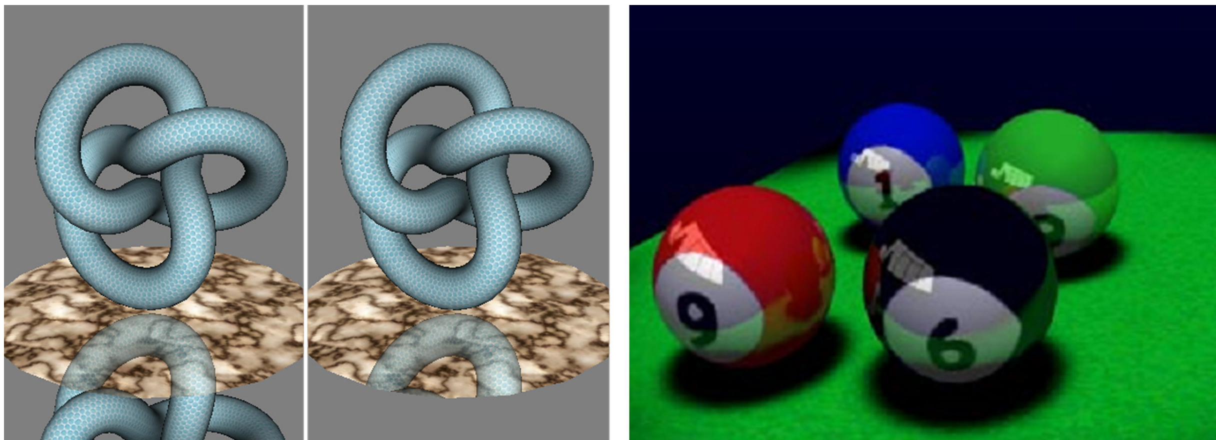


Figure. 17: Left: simple planar reflection achieved by mirroring the object geometry at the reflective plane (left). The stencil buffer is used to mask hidden parts, that are not part of the reflection (right). (left images: [16]) **right:** Interactive reflections on curved objects with geometry transformation technique described by [OR98]

[16] ofps.oreilly.com/static/titles/9780596804824/figs/incoming/WithWithoutStencil.png, [accessed 05/04/2013]

4.2 Image based approaches

The most common image based technique is *environment mapping*, as presented by [BN76]. It is one of the first approximation techniques for specular effects. As shown in figure 18, environment mapping needs a *reference point* to generate a set of textures, as image based approximation of the surrounding geometry. The reflected ray is then used to sample this precomputed map. Often, cube maps are used for environment mapping, meaning that the second step is done using a camera with an aspect ratio of 1.0 and a field of view of 90 degrees to generate six textures, each for every spatial direction, that are texture mapped on the faces of a cube. But there are also environment mapping approaches, that use sphere maps or two hemisphere maps.

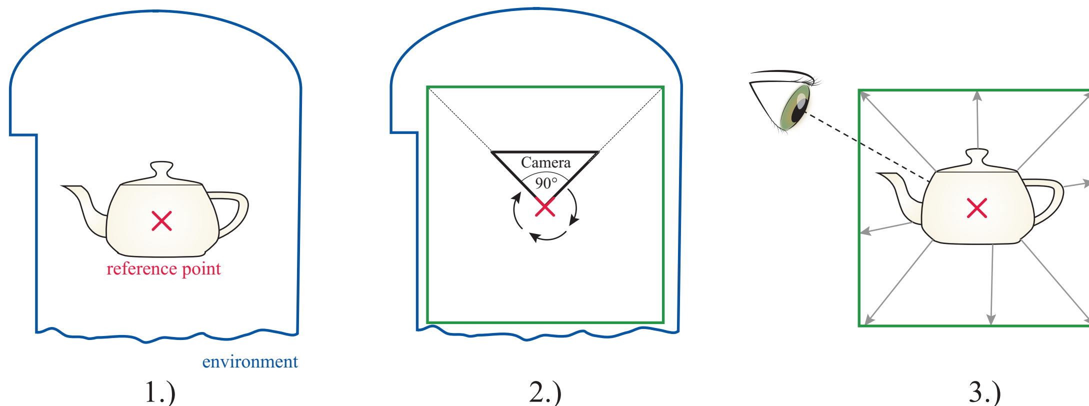


Figure. 18: Environment mapping steps:
1.) Find the center of the object (reference point)
2.) Render an image for every spatial direction
3.) Sample the generated images using the reflected view vector

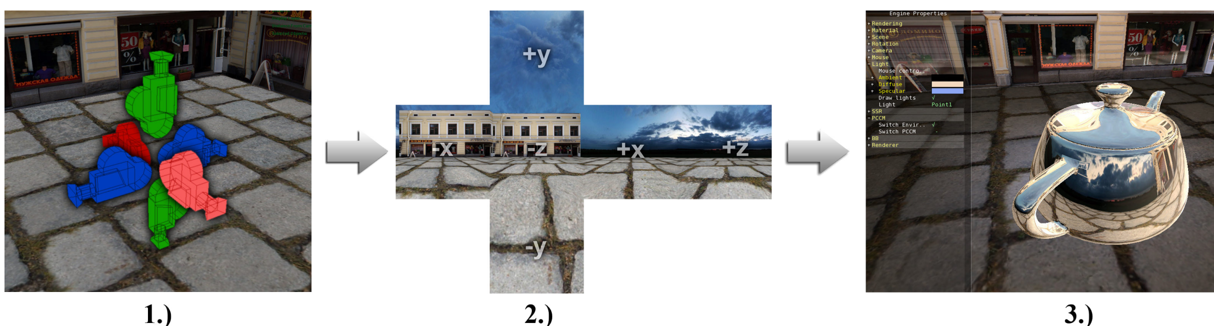


Figure. 19: Environment mapping example:
1.) Cube map generation in *Autodesk Maya*, using one camera for each spatial direction (+x, -x, +y, -y, +z, -z),
2.) Rendered cube map, the texture is passed to the shader using a *samplerCube* uniform,
3.) Final rendering using standard environment mapping

Environment mapping has a major problem: The technique assumes, that the reflected points are infinitely far away and thus cannot present correct parallax when the viewer moves around the object or the object itself moves. Although environment mapping generates plausible reflection effects, because of that problem, the reflecting objects often seem not to be properly fitted into the surrounding scene. Generating new environment maps, if the object moves, the second problem can be easily solved and therefore motion parallax can be provided with additional computation effort and time. Due to it's long-time application, a wide range of extensions for environment mapping exist, that try to enhance the limitations of the original approach. [SKUP$^+$09] gives a great overview of improved environment mapping-based approaches. A common extension of simple environment mapping is prefiltering to achieve glossy reflections. A given reflection model or a BRDF is used to filter the generated environment map, resulting in plausible glossy

reflections when sampling them, but still having the problem of parallax errors. *AMD* released an easy to use cube mapping tool [17][18], that can be used for the prefiltering.

Figure. 20: Left: *AMD CubeMapGen* generates prefiltered environment maps, **right:** Reflections achieved with prefiltered environment maps, right images: [KVHS00]

4.3 Ray tracing geometry

Ray tracing is another classical image synthesis approach. In contrast to the rasterization approach, ray tracing traditionally sends rays per pixel to find an intersection with the world and return the lighted intersected color at the hit point. The approach was not able to generate real-time graphics for a long time, but with increased hardware-performance and research effort, ray tracing algorithms get more and more appliable to real-time.

Algorithm 1: Most plain ray tracing algorithm

input : Screen with amount of $pixels$ $(width \cdot height)$, Surface normals N for each object
output: color C

1 **foreach** $pixel$ **do**
2   compute viewing ray $\overrightarrow{V}$;
3   P = nearest hit point of intersecting V with the world;
4     $C = lighting(P, L, N)$;
5   **return** C;
6 **end**

Traditional ray tracing is implemented on a CPU, suffering from a lack of performance in general. But due to improved hardware, improved intersection algorithms and several acceleration data structures, real-time ray tracing applications are now possible. As a high bandwidth data-parallel computation device, the GPU is well qualified for ray tracing, but even as GPU-ray tracing benefits from CPU-ray tracing research and improvements, not all techniques run equally well on both platforms, most acceleration ideas have to be adapted to the special architecture of the GPU. The traditional rendering pipeline has to be reinterpreted for ray tracing, adapting CPU-algorithms to the GPU's shader programs. [SKUP^{+}09] gives a general overview of possibilities for acceleration structures for implementing a GPU ray tracer and alleges samples of works, that achieve real-time ray tracing by an implementation on the graphics hardware. The other possibility is using an implementation, that dispenses the designated rendering pipeline

[17] developer.amd.com/resources/archive/archived-tools/gpu-tools-archive/cubemapgen/, [accessed 04/04/2013]
[18] code.google.com/p/cubemapgen/, [accessed 04/04/2013]

and it's features and gives access to the high data-parallel stream processing of the GPU, like *CUDA*[19] or *OpenCL*[20] do. Lately, NVidia put a lot of effort in accelerating GPU-raytracing using CUDA by releasing a real-time ray tracing API called *OptiX*[21]. Real-time ray tracing still lacks of speed, changing camera position or light conditions will force an initial tracing, resulting in a noisy result, that will increase in image quality during a small time interval.

Figure. 21: Left: Real-time ray tracing advanced with CUDA, **right:** *Nvidia OptiX* powered GPU-ray tracing (images: [22])

4.4 Ray tracing sampled geometry

Instead of storing original vertex geometry, it can be encoded in some kind of textures. To store three dimensional points, 3D-textures can be used, or the three dimensional information is projected into the two dimensional plane of a texture, which is the basic idea of *geometry images*, described by [CHCH06]. [YWY08] also use *geometry images* to accelerate the ray tracing on the GPU. [PMDS06] generate impostors - billboard impostors and depth map impostors - from the view of the reflecting object to simplify the ray tracing. A similar way is used in the *billboard reflections* technique, which will be covered in section 5.3. *Non-pinhole impostors* ([PHR+09]) eliminate depth artifacts by using an occlusion camera ([MPS05]) for the generation of impostors. *Screen space reflections*, what will be described in the upcoming section 5.4, can also be seen as ray tracing sampled geometry, because a before generated depth-image is sampled using ray marching.

[19] www.nvidia.de/cuda[accessed 04/04/2013]
[20] www.khronos.org/opencl/[accessed 04/04/2013]
[21] developer.nvidia.com/optix, [accessed 05/04/2013]
[22] left: cg.alexandra.dk/2009/08/10/triers-cuda-ray-tracing-tutorial/, [accessed 05/04/2013]
right: www.overclockers.com.ua/video/geforce-gtx480/07-big-nvidia-fermi.jpg, [accessed 05/04/2013]
[23] graphicsrunner.blogspot.de/2008/04/reflections-with-billboard-impostors.html, [accessed 05/04/2013]

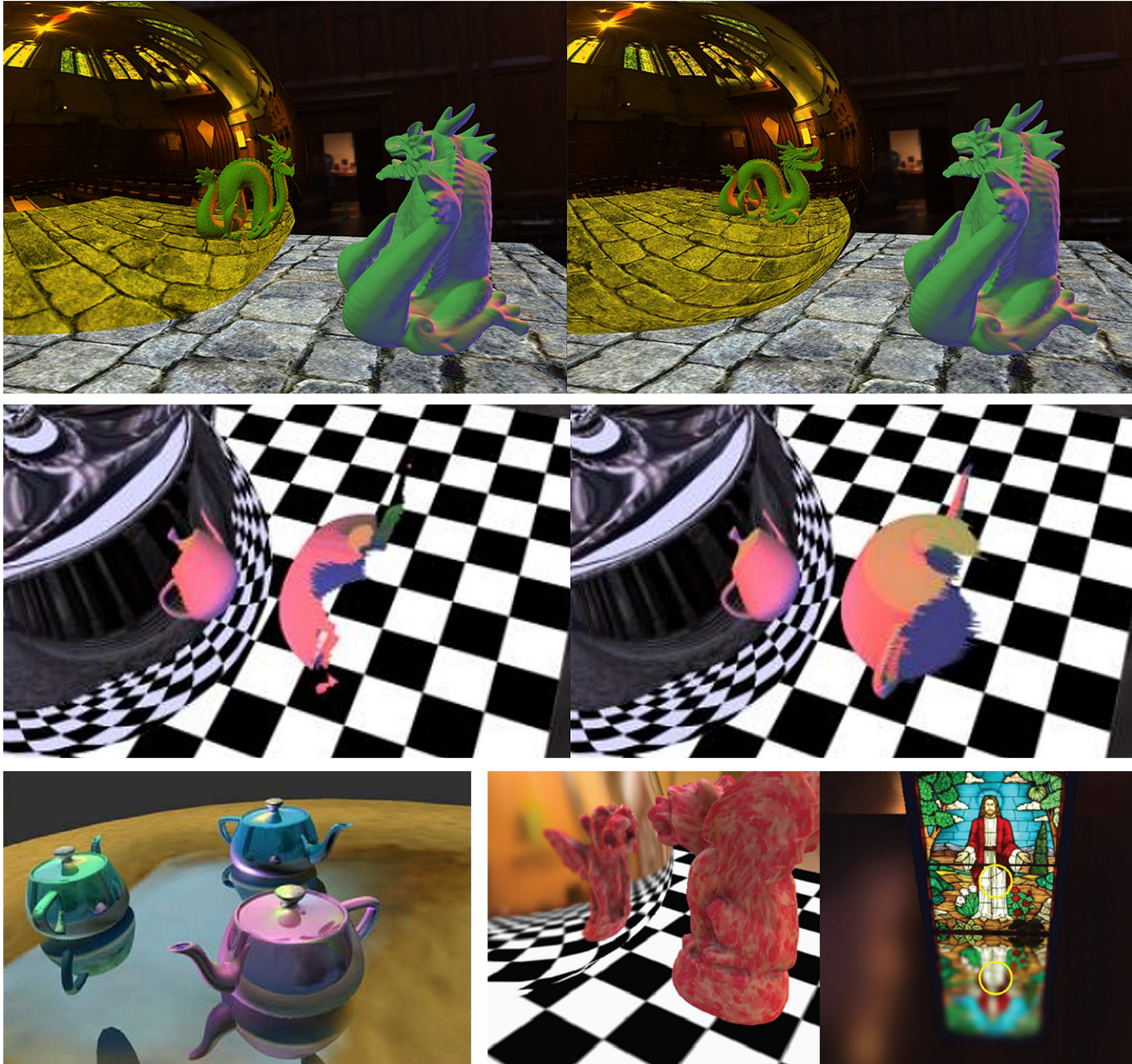


Figure. 22: **Top left:** Environment mapping, **top right:** Impostors used for more accurate reflections, the sphere intersecting the floor quad is more plausible (images [23]), **center:** Samples from a planar pinhole camera (left) and a *occlusion camera* (right), the impostor taken with the occlusion camera covers more of the object (images: [PHR+09]), **bottom left:** Self- and inter-object reflections achieved by GPU ray tracer used with geometry images ([CHCH06]), **bottom right:** results from ray tracer of [YWY08], that was implemented on the GPU using also geometry images

4.5 Hybrid techniques

To get the benefits of both worlds - rasterization pipeline and ray tracing - modern real-time applications such as game engines (e.g. *Unreal Engine 4*or *Cryengine*) combine both techniques. [Sha10] gives a great overview of problems, advantages and disadvantages of this idea and reviews state-of-the-art techniques. [CNS+11b] presents a technique called *Voxel Cone Tracing* that generates plausible visual results in real-time by using a hybrid approach: Primary rays are rendered using the rasterization pipeline, secondary rays are rendered using cone-tracing. The cone-tracing is done on the scene stored in a voxel-based octree, implemented using 3D-textures (described in detail by [Mit12]). This approach is also implemented in the current *Unreal Engine 4* (see figure 23).

[24]left: thumbs.cdn-ec.viddler.com/thumbnail_2_36613e5b_v2.jpg, [accessed 04/04/2013], right: scmods.com/pictures/Unreal-Engine-4-6.jpg, [accessed 04/04/2013]

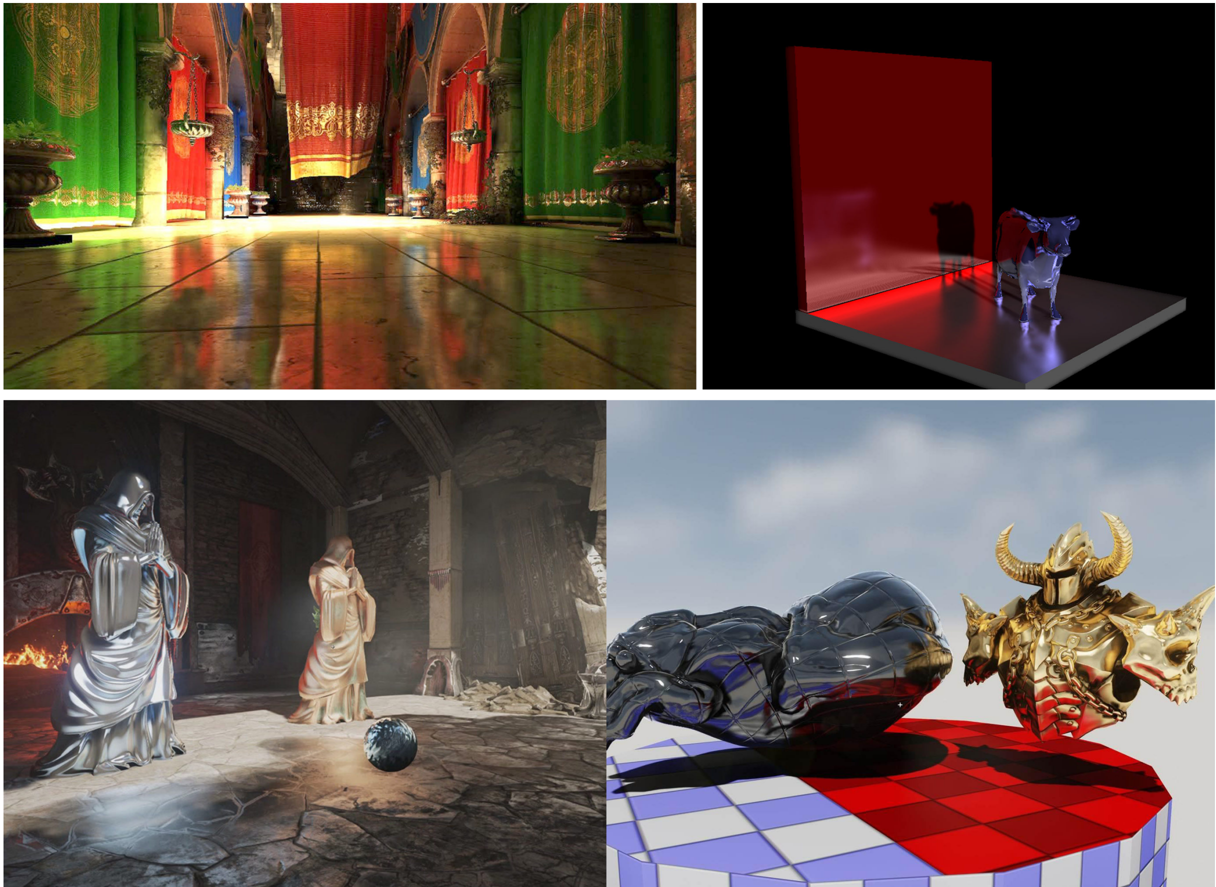


Figure. 23: Top left: Voxel cone tracing result as implemented by [CNS+11b], **Top right:** implementation by computer graphics group at University of Pennsylvania ([IL12]), **Bottom:** Voxel cone tracing as it is implemented in the *Unreal Engine 4*, bottom images: [24]

5 Implementation

5.1 Framework

For implementing the selected choice of algorithms, a framework was created, described in the following. It is based on current *OpenGL* specification (version 4.0+) and *OpenGL Shading Language* (GLSL) and has been written in *C++*. The code can be found at https://github.com/GuidoSchmidt/moge. Figure 24 shows a compact overview of the implemented classes and relationships between them in a simplified UML class diagram.

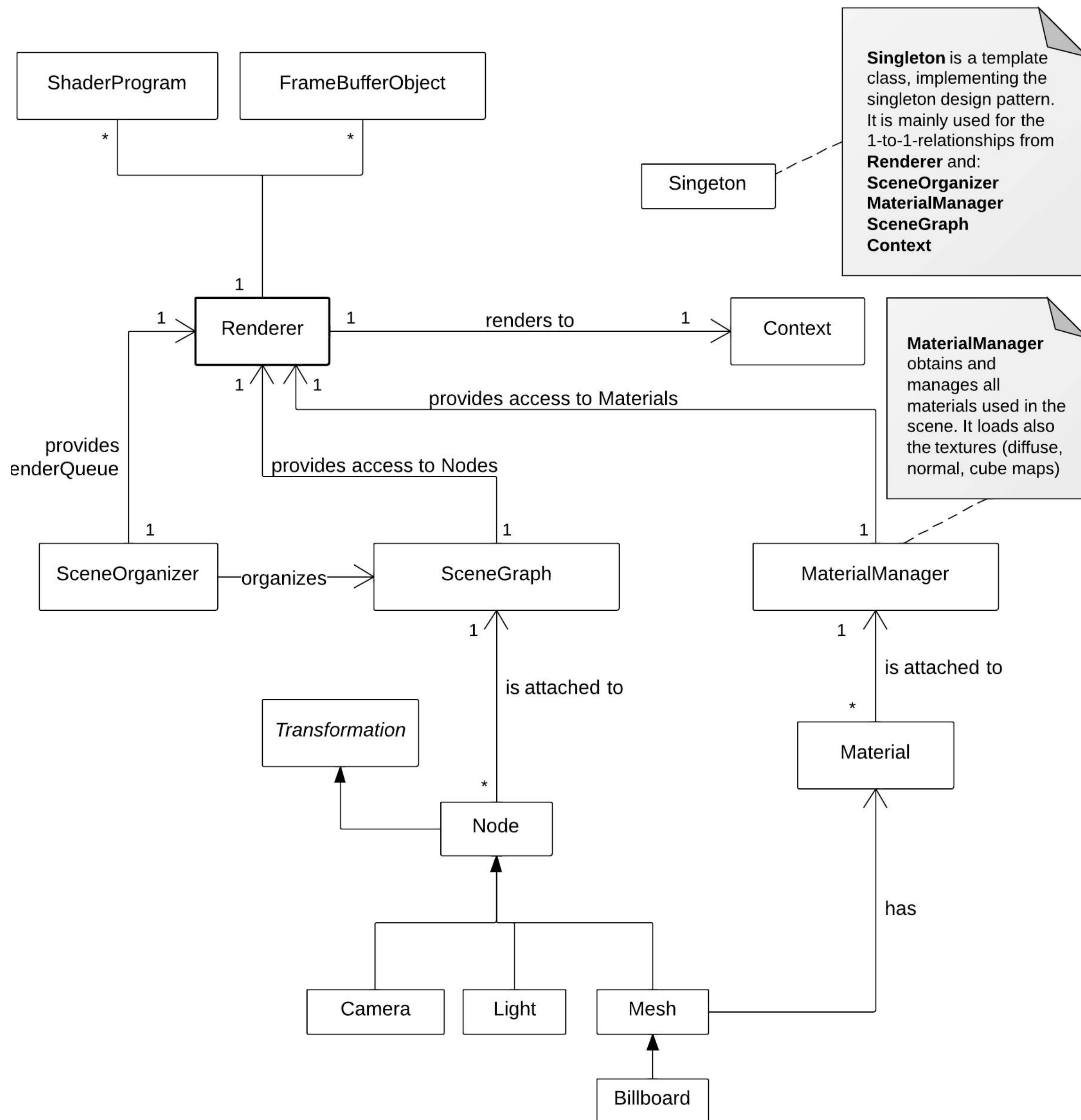


Figure. 24: The implemented framework modelled as UML class diagram

The base class is *Renderer* and runs the draw-function, supplying the rendered images to the OpenGL context, which is created using the *GLFW*[25] library. *GLFW* is also used to load texture files. Textures are

[25] www.glfw.org/, [accessed 15/04/2013]

stored in the *Targa Image Format* (TGA). Besides, the image library *DevIL*[26] is used to load environment map textures. The framework loads two different kind of textures for every object: a diffuse texture with an alpha channel and a normal map texture with a reflective map in the alpha channel.
The shader pipeline is implemented by *Renderer* holding instances of the *ShaderProgram*-class and the *FrameBufferObject*-class, basically being wrappers for *OpenGL*-function calls. *Renderer* is also capable for input-handling such as mouse and keyboard input. The *Renderer*-class also owns singletons of the *SceneGraph*-, the *SceneOrganizer*- and the *MaterialManager*-class, whereby *SceneGraph* is a minimal scene graph implementation consisting of *Nodes*, such as *Mesh*, *Light*, *Camera* and *Billboard*, which is heritage of *Mesh* and will be described more detailed in section 5.3. *SceneOrganizer* is a small class to create a proper render queue, sorted by materials. *MaterialManager* obtains all Materials and loads all textures, that are loaded during the import of a scene. *SceneGraph* also imports scenes in the Collada file-format [27], loading the meshes, appending them to the root-note, loading materials, redirecting texture loadings to the *MaterialManager* and loading light-informations. For importing files into the system, the *Open Asset Import Library*[28] (Assimp) is used. Scenes have been built using *blender* [29], set up and exported to Collada-files using *Autodesk Maya* [30]. For better testing and debugging, a simple *Graphical User Interface* (GUI) was added, employing library *AntTweakBar* [31] (compare figure 25). To get consistent mathematical classes, such as vectors, matrices and also functions, such as the dot-product, that agree with the *GLSL*-Specification, the header-only library *OpenGL Mathematics* [32] (GLM) has been included and used.
The framework is based on a deferred rendering approach, where in a first render pass important properties, such as vertex position, vertex normal, etc. are stored in textures, resulting in a set of sampler textures, that hold the most important data, needed for other computations, e.g. lighting. This first render pass is often called *Geometry Buffer (G-Buffer)*. As pictured in figure 27, the G-Buffer in the implementation creates vertex positions and vertex normals in world space as in view space (or camera space), albedo colors, which are only diffuse colors directly obtained from the textured meshes and reflectance plus the depth, which will be needed later in the *Screen Space Reflections*-computation (see section 5.4). The G-Buffer pass renders all textures into a *Framebuffer Object* (FBO), that stores them, so they can be accessed later from a following render pass, e.g. the lighting pass. The framework has three major algorithms implemented, that can render reflections: *Parallax-corrected Cube Mapping*, based on standard environment mapping using cube maps, *Billboard Reflections* and *Screen Space Reflections*. Figure 26 shows a structural overview of all four render passes, at which point each implemented technique is performed, which output textures are generated by a pass and which textures are obtained by a pre-computed pass. All three render techniques for reflections are described in the following sections 5.2 - 5.4, each in its own section.
The implement framework can load three different testing scenes: a small street scene with two building facades, a car (car model by *dskfnwn*[33]) parked on the street, a cinema entrance and street lanterns. The second one is a museum scene with a dinosaur skeleton (model by Joel Anderson [34]) exhibited in the middle, different paintings, indoor plants and some other small properties. The third is a customized Sibernik cathedral (model from [35]) with stained-glass windows and a wooden floored apsis. For pictures of the scenes, see figure 28). The whole implementation and testing ran on notebook, equipped with a mobile NVIDIA GeForce GT 650M graphics card, that has access to 1024MB video memory.

[26] openil.sourceforge.net/, [accessed 15/04/2013]
[27] www.khronos.org/collada/, [accessed 29/03/2013]
[28] assimp.sourceforge.net, [accessed 29/03/2013]
[29] www.blender.org/, [accessed 15/04/2013]
[30] www.autodesk.de/maya, [accessed 29/03/2013]
[31] anttweakbar.sourceforge.net/, [accessed 29/03/2013]
[32] glm.g-truc.net, [accessed 29/03/2013]
[33] www.turbosquid.com/FullPreview/Index.cfm/ID/411348, [accessed 06/04/2013]
[34] www.creativecrash.com/maya/downloads/character-rigs/c/t-rex-skeleton-rig-from-joel-anderson--2, [accessed 06/04/2013]
[35] http://graphics.cs.williams.edu/data/meshes.xml, [accessed 01/04/2013]

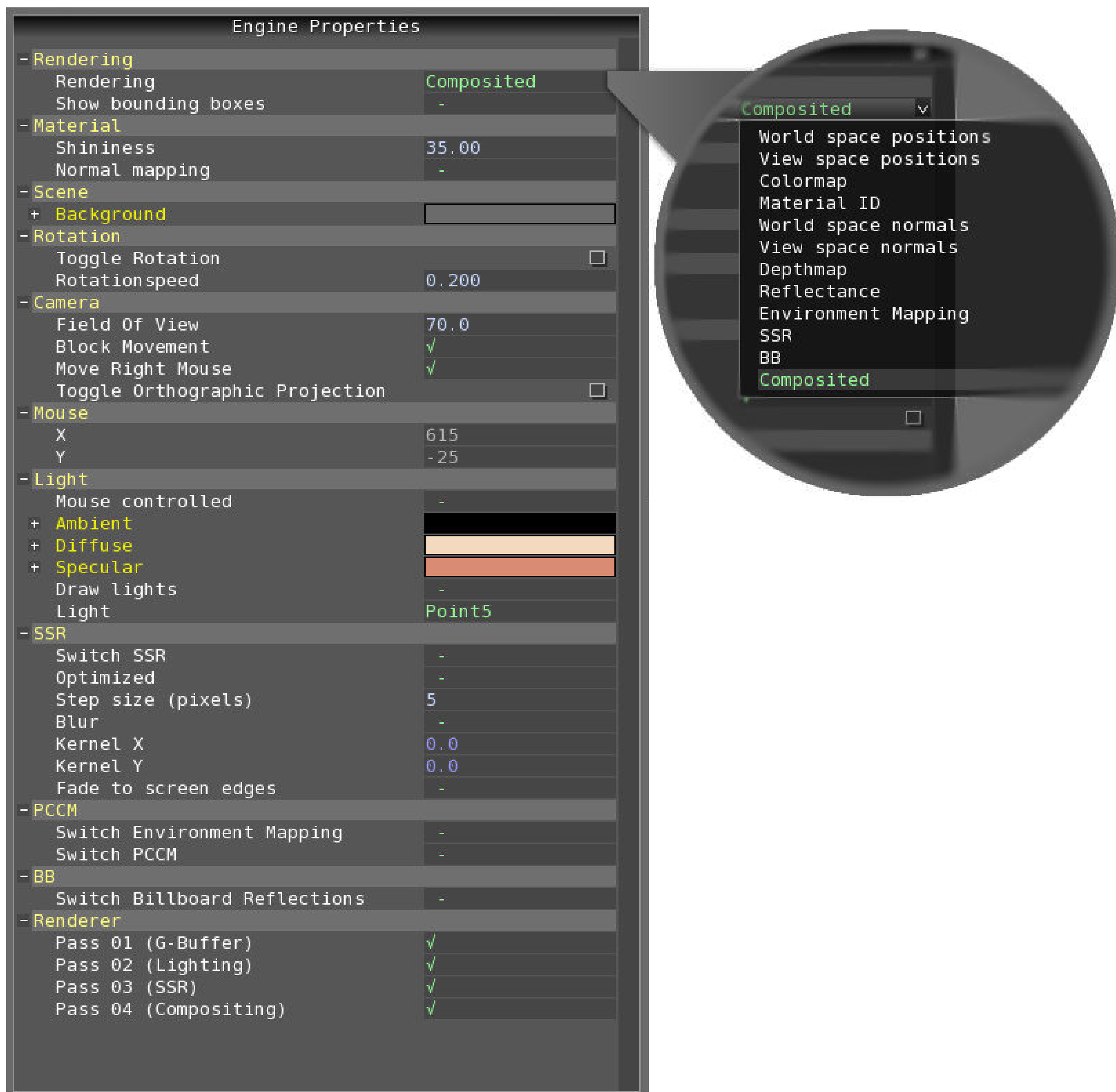


Figure. 25: Implemented GUI, created with library *AntTweakBar*

Table 1: Overview of used test scenes and their properties, texture count excludes cube maps

Scene	Number of vertices	Number of edges	Number of textures	Billboard-count
Street	60226	160590	22	5
Museum	147725	282318	25	3
Cathedral	55020	149719	17	0

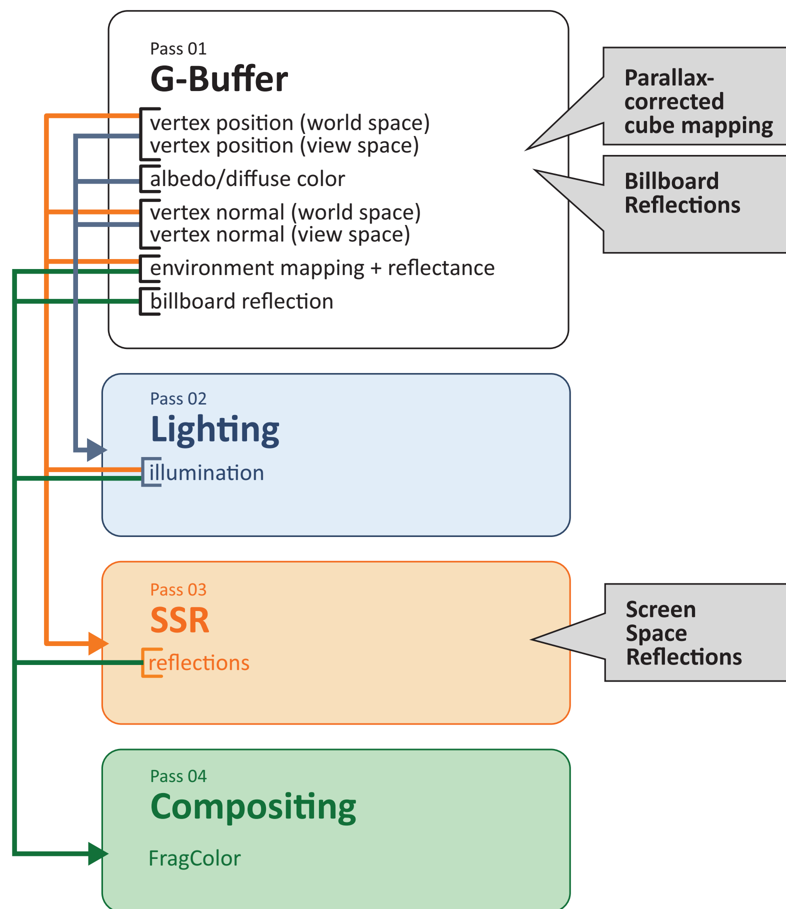
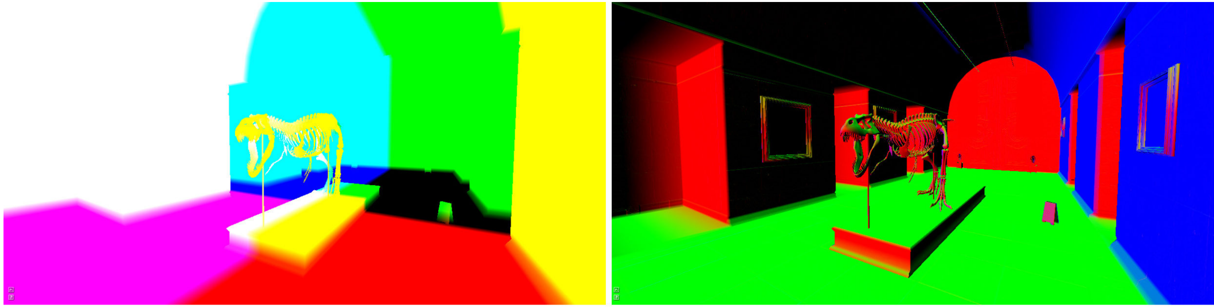
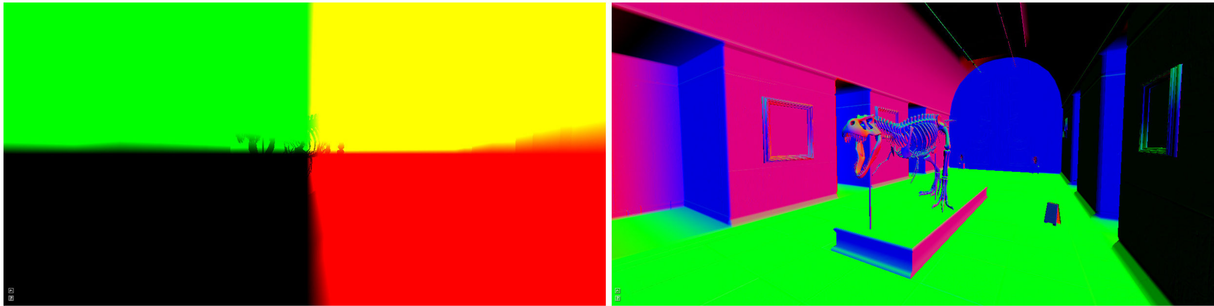


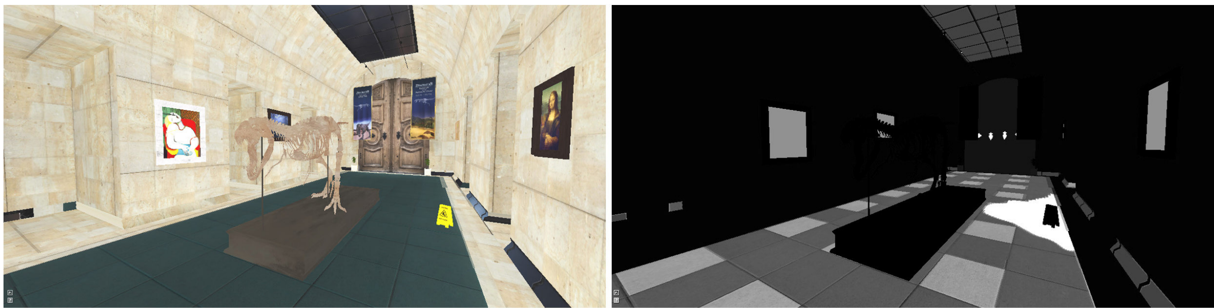
Figure. 26: Structural overview of all four render passes. Properties, written inside of the passes rectangle, describe the output data by a pass, arrows to another pass show the input data for each pass, obtained from an earlier pass. The positions of the computation of the reflection-techniques are illustrated on the right side.



a)



b)



c)

d)

Figure. 27: Images *a) - c)* result from the *G-Buffer* pass
a) left: vertex positions in world space, **right:** vertex normals in world space,
b) left: vertex positions in view space (camera space), **right:** vertex normals in view space (camera space),
c) left: unlighted albedo (diffuse) color, **right:** reflectance (white is high reflective, black is non-reflective),
d) Final result after lighting pass

Figure. 28: a) Simple test scene, **b)** Museum scene, **c)** Sibernik cathedral with custom textures

5.2 Parallax-corrected cube mapping

[SZ12] presents an extended cube mapping approach to get rid of the missing parallax when using cube mapping. By blending local precomputed cube maps together and using proxy box volumes, which are defined before rendering and are used for the intersection with the reflected ray, the view parallax gets corrected. According to Sébastien Lagarde's post on the GPU Pro blog [36], this approach will be also covered in the upcoming book *GPU Pro 4*.

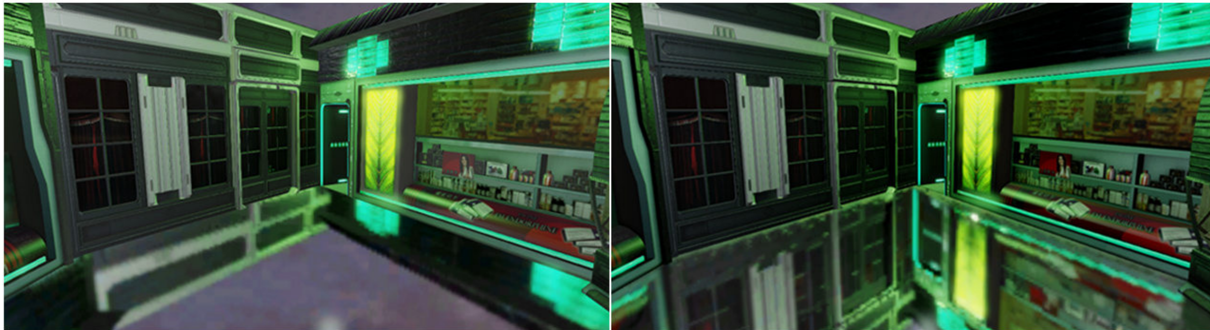


Figure. 29: left: common cube mapping result. **right:** parallax-corrected cube mapping result, upper image: [SZ12] , lower image: [37]

5.2.1 Algorithm

Using *Parallax-corrected cube mapping* (PCCM), a scene with an existing cube map is needed, if it isn't generated while the rendering process. To perform the correction, the algorithm needs an approximation of the local scene, surrounding the local cube map, meaning that the proxy geometry has to positioned and aligned when setting up the scene. The easiest example for a proxy geometry would be an axis aligned bounding box (AABB) of the scene, but also more complex proxy geometries are imaginable. In the described implementation, a cube is used as an AABB, placed and scaled in *Maya* (see figure 30, right). The three dimensional position of the reference point, from where the local cube map was generated, is also needed. The idea behind the parallax-correction is shown in figure 31 and described in algorithm 2.

Algorithm 2: Pseudo code for parallax-corrected cube mapping

---

input : Vertex position P in world space,
Reflected ray R,
cube map position $CM.Position$,
axis aligned bounding box $AABB$

output: Reflected color RC

1 $PlaneIntersectionMax$ = Intersect R with $plane$;
2 $PlaneIntersectionMin$ = Intersect R with $plane$;
3 $FurthestPlaneIntersection = max(PlaneIntersectionMin, PlaneIntersectionMax)$;
4 $Distance$ = GetClosestDistance($FurthestPlaneIntersection$);
5 $Intersection = P + Distance * R$;
6 $CorrectedReflectedRay = Intersection - CM.Position$;
7 $RC = CM.texture(CorrectedReflectedRay)$;
8 **return** RC;

---

[36] http://gpupro.blogspot.de/
[37] http://seblagarde.files.wordpress.com/2012/08/parallax_corrected_cubemap-gameconnection2012.pdf
[38] http://seblagarde.files.wordpress.com/2012/08/parallax_corrected_cubemap-gameconnection2012.pdf

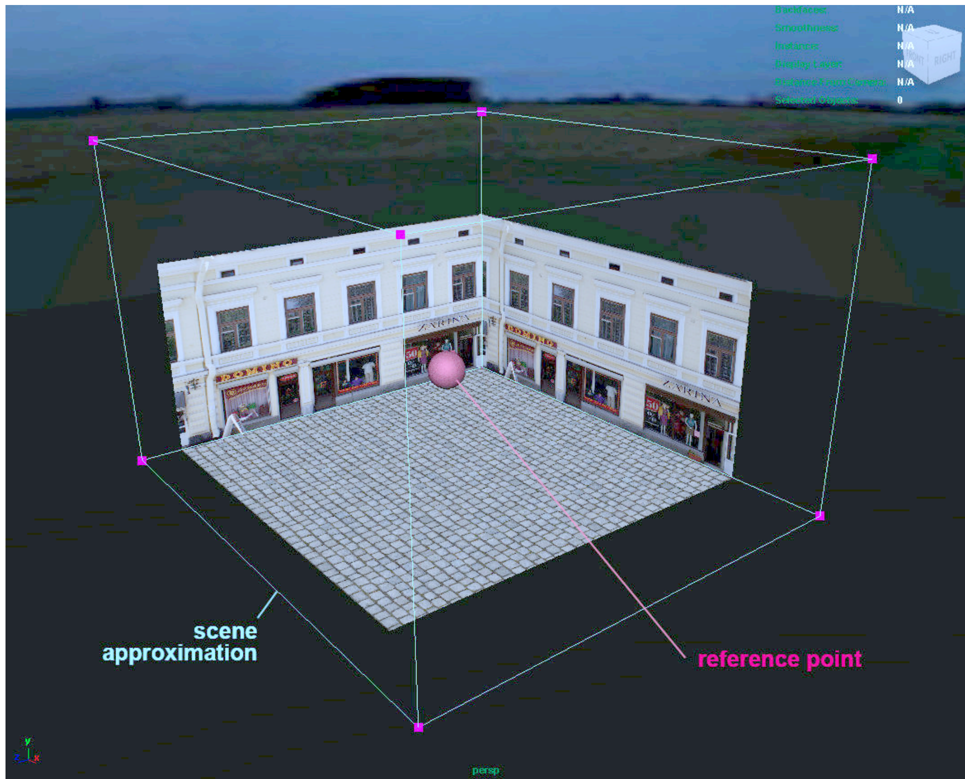


Figure. 30: AABB positioning in *Maya*, the red ball is the reference point from where the cube map was generated

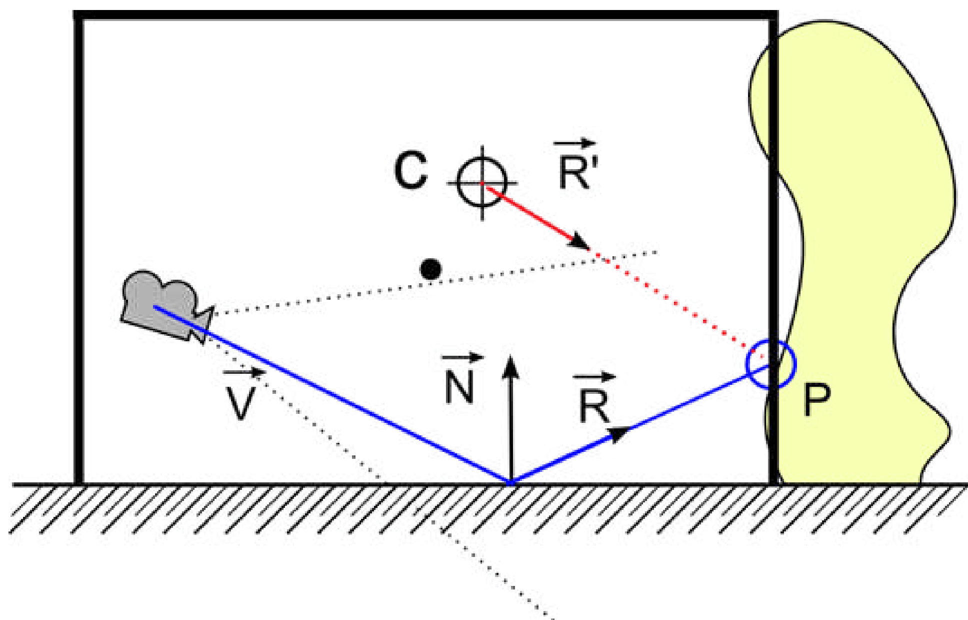


Figure. 31: Parallax-correction: $\overrightarrow{V}$ is the view vector, $\overrightarrow{N}$ is the vertex normal, $\overrightarrow{R}$ is the reflection vector, $\overrightarrow{R'}$ is the corrected reflection vector, P is the intersection point with the proxy geometry (image: [38])

The implementation can be done in the G-Buffer pass, since there is no precomputed data needed. The cube map is generated in *Maya* and passed as a uniform parameter to the fragment shader. The complete parallax-correction function ***ParallaxCorrecteCubeMapping(in vec3 wsPosition, in vec3 wsReflectionVector)*** can be found in listings 1 on page 26. As input parameters the function takes the vertex position and the computed reflection vector, both in world space. First, the intersection with the proxy geometry is computed for the maximum and the minimum of the AABB. Afterwards, the plane, which is furthest away has to be found by using GLSL's maximum-function on both plane intersections. To use the reflected ray for sampling the cube map, the distance to that plane has to be found. This is done by finding the shortest distance of every component of the furthest plane, as done in line 11. In line 13 of listings 1, the corrected intersection point is computed based on the parametric ray equation from equation 5. Then the reflected ray can be corrected and the used to sample the cube map, as done in line 20.

Listing 1: Shader code for Parallax-corrected cube mapping

```
1  //  Parallax corrected cube mapping
2  vec3 ParallaxCorrecteCubeMapping(in vec3 wsPosition, in vec3 wsReflectionVector)
3  {
4    // Intersect with first AABB plane
5    vec3 firstPlaneIntersection  = (AABB.max - wsPosition) / wsReflectionVector;
6    // Intersect with second AABB plane
7    vec3 secondPlaneIntersection = (AABB.min - wsPosition) / wsReflectionVector;
8
9    // Intersect with second AABB plane
10   vec3 furthesPlane = max(firstPlaneIntersection, secondPlaneIntersection);
11   float distance = min( min(furthesPlane.x, furthesPlane.y), furthesPlane.z );
12
13   vec3 wsIntersection = wsPosition + distance * wsReflectionVector;
14
15   // Calculate parallax-corrected reflection vector
16   // wsCubeMapPosition is the position from where the cube map was generated (reference point)
17   vec3 wsParallaxCorrectedReflectionVector = wsIntersection - wsCubeMapPosition;
18
19   // Sample cube map texture with parallax-corrected vector
20   vec3 reflectedColor = CubeMapping(wsParallaxCorrectedReflectionVector);
21
22   return reflectedColor;
23 }
```

5.2.2 Results

PCCM provides a good possibility for enhancing the common environment mapping technique. It suits best for narrow rooms, such as corridors, halls or alleys (see figure 32. As the classical environment mapping technique, PCCM can not provide good results for object inside of the scene, if they do not get their own local environment map. For example the pillars in the cathedral scene are not reflected correctly, as illustrated by figure 33, because a simple AABB proxy geometry doesn't model their position correctly. More complex proxies would be needed, complicating the intersection with the reflected ray.
Real-time applicability of PCCM is given by default, because the overhead for PPCM is not excessively more than for standard environment techniques. Only correcting the reflected ray by intersecting it with the proxy geometry does not affect the performance in an overstated way, as long as the original environment mapping techniques does provide real-time framerates. The PCCM, that was implemented in the above described framework runs at average rates of roughly 60 fps at a resolution of 1600 to 900 pixels.

Figure. 32: Results for PCCM tested with the street scene, **left:** PCCM without normal mapping: inner-scene objects, such as the car, do not have reflections, **right:** close-up with normal mapping

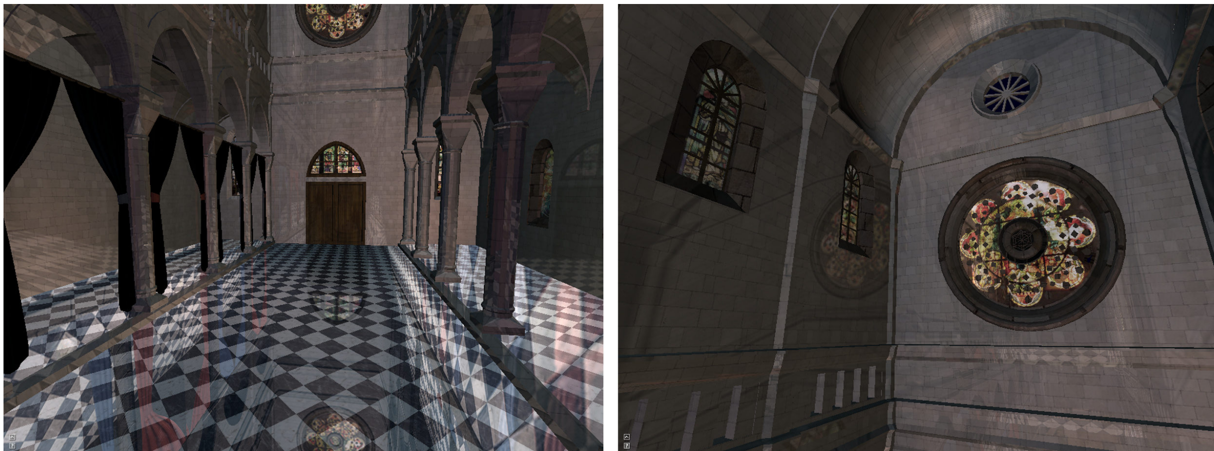


Figure. 33: Results for PCCM tested with the cathedral scene, **left:** due to the simplicity of an AABB proxy geometry, the pillars and the walls behind them are not reflected correctly, **right:** close-up of the clerestory and the rosette window: small parallax errors occur, due to the proxy being modelled at the width of the center nave)

5.3 Billboard reflections

Billboard reflections implement the idea of using textured impostor quads for objects, therefore called *image based reflections* in epicgames's Unreal Engine [39]), which will simplify the computation of their reflections. The reflected ray can easily be intersected with the given quad and common algorithms from ray tracing can be adapted for that purpose. They best suit for high contrast reflections, that fit simple light emitting objects like neon signs in an urban environment, as they were mostly used in Unreal Engine 3 tech demo *Samaritan* (see figure 34).

Figure. 34: Billboard reflections in Unreal Engine 3 tech demo *Samaritan* (images: [40])

5.3.1 Algorithm

First, artists define simple planes, consisting of four vertices, that are arbitrary moved, rotated or scaled in world space. The plane is textured with an RGBA-image, meaning it can have an alpha map for blending out pixels. The scene is set up using *Autodesk Maya* and billboard meshes are tagged with the name-prefix 'Billboard', to identify them in the import-process, where Billboard-objects are created and appended to the scenegraph.

Figure. 35: a) Billboard placement in *Maya*, b) RGB-texture of the billboard, c) Alpha map of the billboard

Reflections are calculated by intersecting the reflected ray with the billboard mesh in the shader program of the G-Buffer. Model-matrices and textures of all billboards in the scene are passed as uniform arrays to the shader. The basic functionality of the billboard reflection algorithm works as follows:

[39] http://udn.epicgames.com/Three/ImageBasedReflections.html

[40] http://udn.epicgames.com/Three/ImageBasedReflections.html

Figure. 36: Setup of billboard quads for the street scene in *Maya*

Algorithm 3: Pseudo code for the billboard reflections algorithm

input : $standardTriangle$,
billboard = $(billboard.texture, billboard.modelMatrix)$,
reflected ray R
output: Reflected color RC

```
1 foreach billboard do
2     foreach vertex of standardTriangle do
3         Multiply vertex with billboard.modelMatrix;
4     end
5     uv = Intersect R with standardTriangle;
6     RC = billboard.texture(uv);
7     return RC;
8 end
```

To find the intersection of the reflected ray and the billboard, the shader program uses a standard triangle, that lies in the xz-axis, with vertices (see figure 37):

$V_0 = (0.5, 0.0, 0.5)$, $V_1 = (-0.5, 0.0, -0.5)$, $V_2 = (-0.5, 0.0, 0.5)$

Calculating the point where the ray hits the billboard, the ray-triangle intersection algorithm from [MT97] is used. The intersection of the reflected ray and the quad can be seen as the following equation:

$$\vec{R}(t) = T(u, v) \tag{6}$$

$$\vec{R}(t) = O + t \cdot \vec{D} \tag{7}$$

$$T(u, v) = (1 - u - v) \cdot V_0 + u \cdot V_1 + v \cdot V_2 \tag{8}$$

Where $\vec{R}(t)$ is the parametric ray equation from section 3 and $T(u, v)$ defines the barycentric coordinates on the billboard: whereby (u, v) must fulfill the the following conditions:

$$u \geq 0, v \geq 0 \tag{9}$$

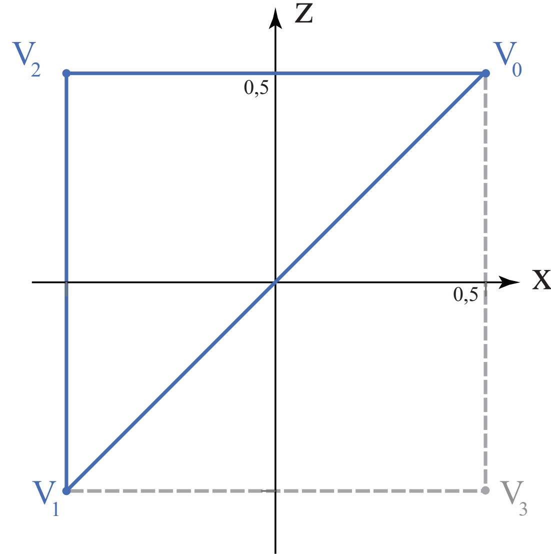


Figure. 37: Standard billboard triangle used to calculate the intersection between reflected ray and billboard in the shader program

(u, v) can directly used for texture mapping the billboard. Barycentric coordinates on a triangle would require (u, v) to fulfill the additional condition $u + v \leq 1$, but as the billboards are simple quads, the conditions from (9) are sufficient.
Insertion of equations 7 and 8 into equation 6 results in:

$$O + t \cdot \overrightarrow{D} = (1 - u - v) \cdot V_0 + u \cdot V_1 + v \cdot V_2 \tag{10}$$

Now rearranging the terms gives:

$$O - V_0 = -t \cdot D + u \cdot (V_1 - V_0) + v \cdot (V_2 - V_0) \tag{11}$$

$$\Leftrightarrow \begin{pmatrix} -D, & V_1 - V_0, & V_2 - V_0 \end{pmatrix} \begin{pmatrix} t \\ u \\ v \end{pmatrix} = O - V_0 \tag{12}$$

To get parameters (u, v, t), the linear system of this equation has to be solved. The geometric idea behind that approach is to translate the triangle to the origin, transforming it to a unified triangle, lying in the xz-axis and translating the ray, so that it is aligned along the y-axis (see figure 38).

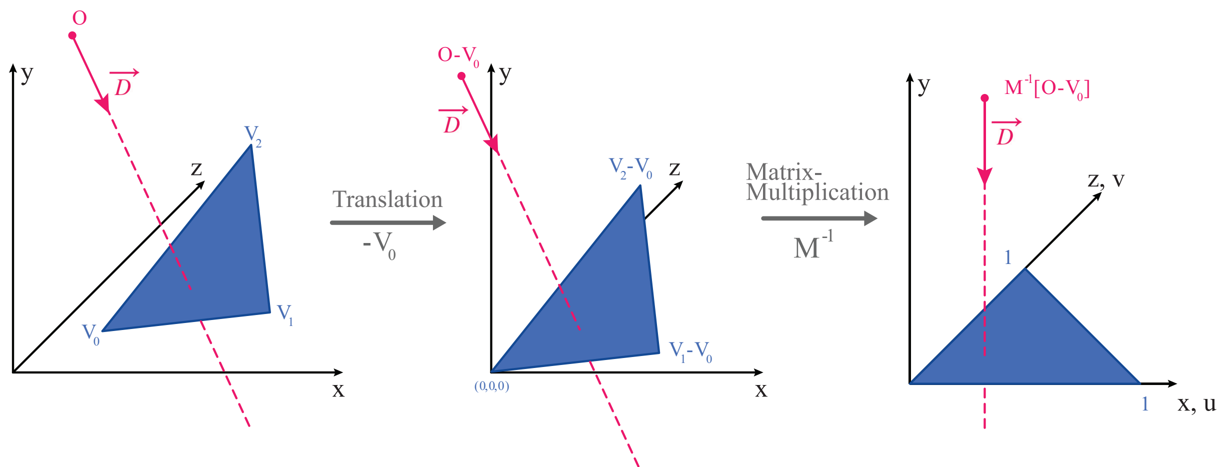


Figure. 38: Geometric idea of the ray-triangle intersection algorithm, $M = \begin{pmatrix} -D, & V_1 - V_0, & V_2 - V_0 \end{pmatrix}$ is the matrix from equation 12

As [MT97], Cramer's rule is used to obtain the solution to equation 12, ending up with:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{|\ -D,\ E_0,\ E_1\ |} \begin{pmatrix} |T,\ E_0,\ E_1| \\ |-D,\ T,\ E_1| \\ |-D,\ E_0,\ T| \end{pmatrix} \tag{13}$$

Whereby $E_0 = V_1 - V_0$ is the edge between V_0 and V_1 and $E_1 = V_2 - V_0$ is the edge between V_0 and V_2. Using formula $|\left(\ A, B, C\ \right) = -(A \times C) \bullet B = -(C \times B) \bullet A|$ ($\times$ is the cross product, $\bullet$ is the dot product) we can substitute into $P = D \times E_2$ and $Q = T \times E_1$ to get the final equation

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{P \bullet E_1} \begin{pmatrix} Q \bullet E_2 \\ P \bullet T \\ Q \bullet D \end{pmatrix} \tag{14}$$

Thereby factors P, Q and T can be reused in the implementation.

The implemented shader code is directly adapted from the mathematical approach as shown in listing 2. The main function is ***BillboardReflections(in vec3 wsPosition, in vec3 wsReflectVec)*** (see listings 2, page32), which takes the ray's origin and destination as parameters and returns the reflected color. For the intersection of every billboard inside of this function, function ***IntersectTriangle(in vec3 rayOrigin, in vec3 rayDirect, in vec3 vert0, in vec3 vert1, in vec3 vert2)*** (see listings 3, page 32) is called. The intersection-function also takes the ray - origin and destination - as the first two parameters, followed by the three vertices, that define the billboard. The vertices refer to the standard triangle, known from 37 and are hard coded into the ***BillboardReflections()***-function (listings 2, line 11-14). They are transformed with the model matrix for each billboard. ***IntersectTriangle()*** first creates two edges of the associated triangle. In line 21, P is calculated and in the following lines 25 and 26 used to create the determinant and its inverse. By calculating T and Q, finally the uv-coordinates (u, v) and the distance t can be calculated and returned as the function's result. After obtaining the coordinates, they are checked to be inside of *texture space* and the alpha test is passed. The reflected billboard-color is then returned.

Listing 2: Shader code for billboard reflections

```
1  // Billboard reflections
2  vec4 BillboardReflections(in vec3 wsPosition, in vec3 wsReflectVec)
3  {
4    // Reflected color, that will be returned
5    vec4 reflectedColor = vec4(0.0);
6
7    // For every Billboard in the scene
8    // billboardCount is passed as uniform parameter
9    for(int i = 0; i < billboardCount; i++)
10   {
11     // Initial billboard (triangle is sufficient)
12     vec3 vert0 = vec3( 0.5, 0.0,  0.5);
13     vec3 vert1 = vec3(-0.5, 0.0, -0.5);
14     vec3 vert2 = vec3(-0.5, 0.0,  0.5);
15
16     // Build reflected ray
17     // ray origin is the current vertex position in world space
18     // ray direction is the at the normal reflected view vector
19     vec3 rayOrigin = wsPosition;
20     vec3 rayDirect = wsReflectVec;
21
22     // Transform billboard to world space
23     // Each vertex of the initial billboard is transformed into the object space
24     // of the current billboard
25     vert0 = ( Impostor[i].ModelMatrix * vec4(vert0, 1.0) ).xyz;
26     vert1 = ( Impostor[i].ModelMatrix * vec4(vert1, 1.0) ).xyz;
27     vert2 = ( Impostor[i].ModelMatrix * vec4(vert2, 1.0) ).xyz;
28
29     // Intersect the billboard and get texture coordinates
30     // (u,v) are the texture coordinates on the triangle / billboard
31     // t is the distance
32     vec3 uvt = IntersectTriangle(rayOrigin, rayDirect, vert2, vert0, vert1);
33
34     // Break, if the distance if lower than 0.001
35     // this would be the real billboard, not a reflected one
36     if(uvt.z <= 0.001)
37       break;
38
39     // Check if texture coordinates are valid between 0.0 and 1.0
40     // Because billboards are quads, these two conditions are sufficient
41     // (Triangle intersection would have the additional condition: (uvt.x + uvt.y) <= 1.0 )
42     if(uvt.x > 0.0 && uvt.x < 1.0 &&
43        uvt.y > 0.0 && uvt.y < 1.0)
44     {
45       // Check alpha channel of billboard texture
46       float alpha = texture(ImpostorTex[i], uvt.xy).a;
47       // Get texture for billboard and do alpha blending
48       reflectedColor.rgb = texture(ImpostorTex[i], uvt.xy).rgb * alpha;
49     }
50   }
51
52   return reflectedColor;
53 }
```

Listing 3: Shader code for ray-billboard intersection

```
// Intersect a triangle with a given ray
// Used here to Intersect billboard quads
vec3 IntersectTriangle(in vec3 rayOrigin, in vec3 rayDirect, in vec3 vert0, in vec3 vert1, in vec3↩
    vert2)
{
  // Parameters
  // (u, v) are the barycentric coordinates
  // t is the distance
  float t, u, v;

  // Triangle edges
  vec3 edge0, edge1;

  // Determinant and it's inverse
  float det, inv_det;

  // Calculate edges
  edge0 = vert1 - vert0;
  edge1 = vert2 - vert0;

  // Calculate p vector (P = D x E_2)
  vec3 pvec = cross(rayDirect, edge1);

  // Calculate determinant
  // and inverse of determinant
  det = dot(edge0, pvec);
  inv_det = 1.0 / det;

  // Calculate t vector (T = O - V_0)
  vec3 tvec = rayOrigin - vert0;
  // Calculate u parameter
  u = dot(tvec, pvec) * inv_det;

  // Calculate q vector (Q = T x E_0)
  vec3 qvec = cross(tvec, edge0);
  // Calculate v parameter
  v = dot(rayDirect, qvec) * inv_det;

  // Calculate distance
  t = dot(edge1, qvec) * inv_det;

  // Return results
  return vec3(u, v, t);
}
```

5.3.2 Results

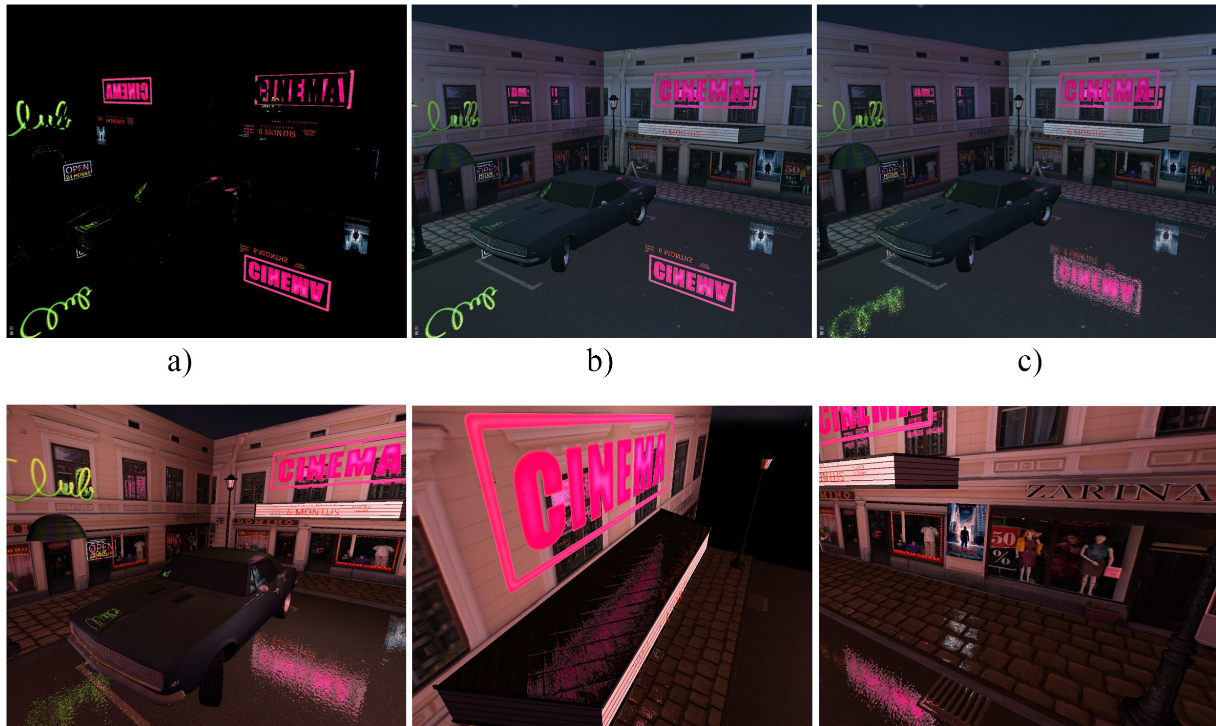


Figure. 39: Billboard reflections in the screet scene
Top: a) Only billboard reflections, **b)** billboard reflections without normal mapping, **c)** results with normal mapping
Bottom: Some detailed views of billboard reflections

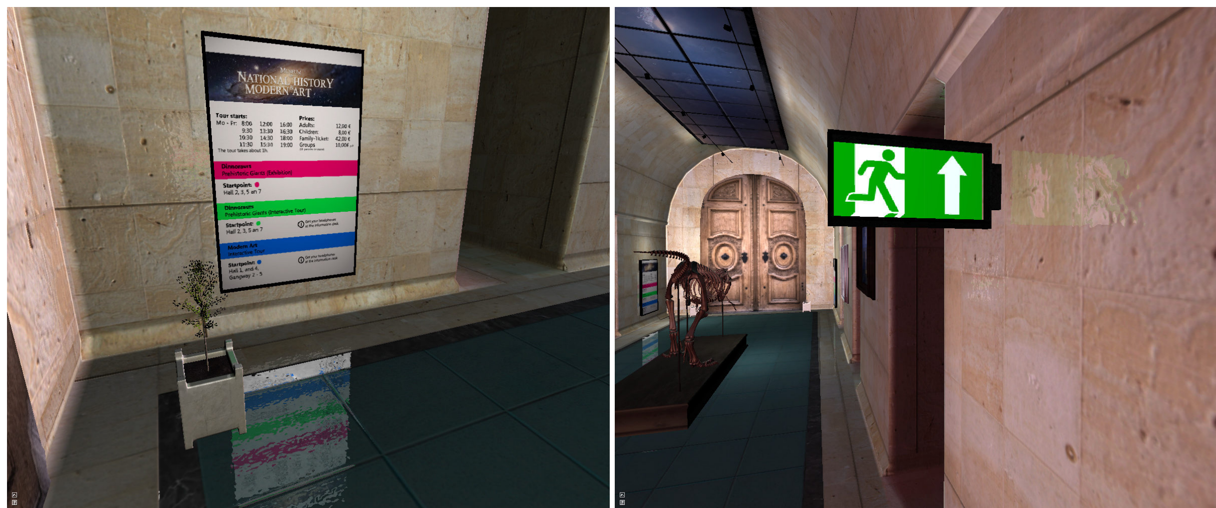


Figure. 40: Billboard reflections in the museum scene, mark the reflection of the exit sign on the wall

Real objects, that are generally shaped like quads, for example billboard advertisement, illuminated posters or street signs can be intuitively models with billboard quads. Their reflections have a significant handicap: they only work for the placed quads, but for this purpose they create convincing results and run at an excellent performance. They best suit for high contrast reflections, e.g. reflections caused by strong lights like neon advertisements in cities at night time, as seen in section 2.2. Another major issue of BBR is missing occlusion. When creating reflections, it is not tested, whether another object occludes the billboard on the reflected ray. Thereby reflections occur at positions, where they are not expected to be, as the examples in figure 41 illustrates. Evaluating the performance, the chart in figure 42 shows the results of measuring frames per second over one minute of time for BBR. The zero values at the

beginning of the measurement represent the loading time of a scene. While recording, the camera was moved through the scene constantly.

Figure. 41: Missing occlusion for the reflection on the street. The reflected ray should be blocked by the parked car.

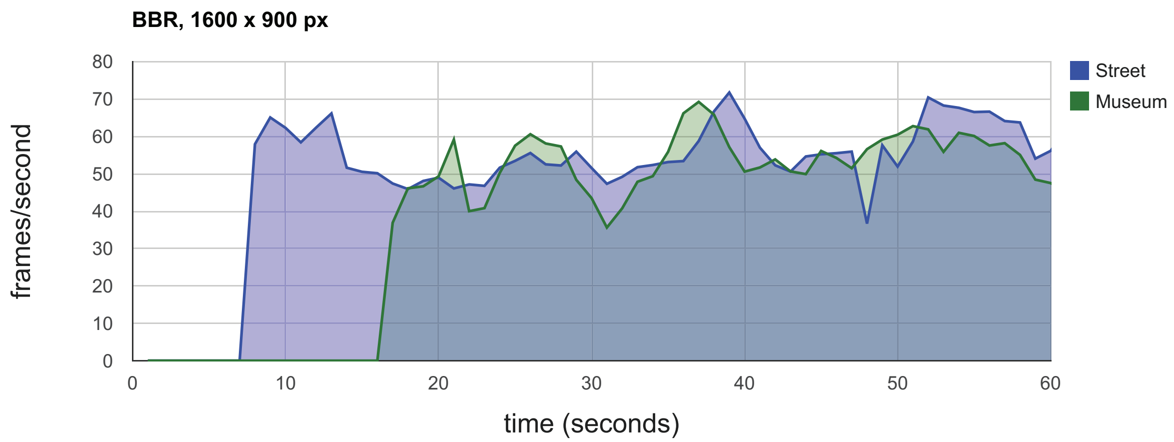


Figure. 42: Comparison of the frames per second for the street and the museum scene (there are no billboards placed in the cathedral scene)

5.4 Screen space reflections

Before explaining *Screen Space Reflections*, first the terms *screen space* and *texture space* are described in detail to understand their difference:

> *screen space* describes the space after clipping and after the perspective projection, where the range of the canonic volume lies in between the bounds of the interval $[1.0, -1.0]$ in every spatial direction.
>
> *texture space* describes the bounds of a texture: A two dimensional space in range from 0.0 to 1.0, often sampled using *uv-coordinates*, as used for texture mapping. Depth in *texture space* can only be inquired by sampling the *depth buffer* or a depth texture, written by a previous render pass.

Screen space approaches make use of the deferred rendering technique, doing calculations only on the data, that was stored in textures by the G-Buffer pass. Therefore calculations are only done, where the viewer can see something, which can save computation time, but may result in lack of visual details, especially when it comes to reflections. An overview of other screen space approaches and their advantages and disadvantages can be found in [Wil].

5.4.1 Algorithm

Screen space reflections utilize the results of the G-Buffer to calculate the reflections. The reflection vector is calculated in view space in the fragment shader for each pixel, using position- and normal-information from the textures of the G-Buffer. The reflection vector is then projected into screen space using the camera's projection matrix (the projection will be described more detailed in the following section 5.4.1). Ray marching along the screen space reflection vector is done afterwards, sampling the depth texture at the ray's position and comparing the sampled depth with the ray's z-value: If the ray's depth is bigger than the sampled depth, the ray has hit something and the reflected color can be returned at that position, sampling it from the texture, obtained from the lighting pass.

Algorithm 4: Pseudo code of the screen space reflections algorithm

input : Reflected ray R,
Depth buffer texture $DepthTex$ (G-Buffer),
Diffuse texture $DiffuseTex$ (lighting pass),
output: Reflected color RC

```
1 ssR = project R into screen space;
2 while Raymarch along ssR do
3 |  if ssR.z > DepthTex at position ssR.xy) then
4 |  |  RC = DiffuseTex at position ssR.xy);
5 |  end
6 end
7 return RC;
```

In OpenGL a matrix is needed to project a three dimensional point onto the near plane of the camera: The projection matrix. In the following, the mathematics behind the projection will be covered, because understanding the projection is an essential part for screen space algorithms. When the camera is defined, it has a view frustum, resulting from its opening angle, its aspect ratio and the near and far plane, set by *GLM*'s *glm::perspective()*-function. The view frustum is generated by four planes, described by six values: $near(n)$ and $far(f)$, $left(l)$ and $right(r)$ and $top(t)$ and $bottom(b)$ as shown in figure 43. Constructing this projection matrix, it can be started with the xy-coordinates of a point: With the intercept theorem, the xy-coordinates of a point $P_{vs} = (x_{vs}, y_{vs}, z_{vs})$ in view space (vs) are projected to a point

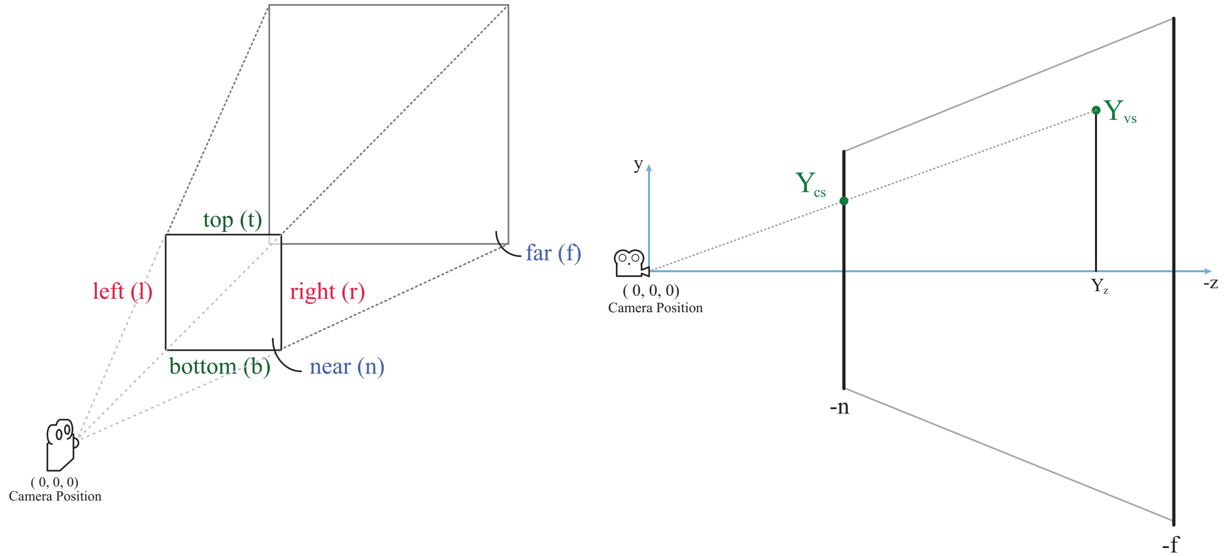


Figure. 43: left: View frustum of the camera as it is used in OpenGL, **right:** Projection of a the y-component of a three dimensional point onto the near plane of the camera's frustum

$P_{cs} = (x_{cs}, y_{cs}, z_{cs})$ in clip space (cs) (see figure 43, right):

$$\frac{x_{cs}}{x_{vs}} = \frac{-n}{z_{vs}} \Leftrightarrow x_{cs} = -\frac{n \cdot x_{vs}}{z_{vs}} \tag{15}$$

$$\frac{y_{cs}}{y_{vs}} = \frac{-n}{y_{vs}} \Leftrightarrow y_{cs} = -\frac{n \cdot y_{vs}}{z_{vs}} \tag{16}$$

After the projection into clip space, the coordinates are still homogeneous coordinates and need to be transformed into normalized device (ndc) coordinates. To do this, the clip coordinates are divided by their w-component, which is set to $-z_{vs}$, so the fourth row of the projection can be written as $(0, 0, -1, 0)$:

$$M_{projection} = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ y_0 & y_1 & y_2 & y_3 \\ z_0 & z_1 & z_2 & z_3 \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{17}$$

The clip coordinates now have to be normalized to suit range $[-1, 1]$. Using 18 and projecting x_{ndc} onto 1 and x_{cs} onto r, the result is:

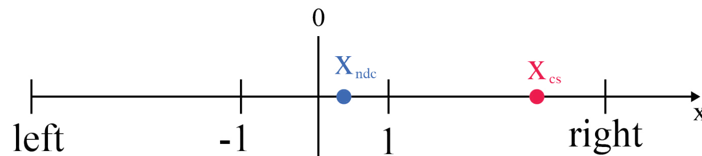


Figure. 44

$$x_{ndc} = \frac{1-(-1)}{r-l} \cdot x_{cs} + \beta \Leftrightarrow 1 = \frac{2 \cdot r}{r-l} + \beta \Leftrightarrow \beta = -\frac{r+l}{r-l} \tag{18}$$

$$\Leftrightarrow x_{ndc} = \frac{2 \cdot x_{cs}}{r-l} - \beta \Leftrightarrow x_{ndc} = \frac{2 \cdot x_{cs}}{r-l} - \frac{r+l}{r-l} \tag{19}$$

Now the coordinate from 15 can be inserted into the equation for x_{ndc} from 20, resulting in:

$$\Leftrightarrow x_{ndc} = \frac{\left(\frac{2 \cdot n}{r-l} \cdot x_{vs} + \frac{r+l}{r-l} \cdot z_{vs}\right)}{-z_{vs}} \tag{20}$$

Whereby the numerator gives x_{cs}. The whole calculation is done analogical to the y-coordinate by replacing r with t and l with b, resulting in a projection matrix, where first and second rows are:

$$M_{projection} = \begin{pmatrix} \frac{2\cdot n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2\cdot n}{t-b} & \frac{t+b}{t-b} & 0 \\ z_0 & z_1 & z_2 & z_3 \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{21}$$

The only missing row now is the third, defining the z-coordinate of the projected point. The problem here is, that z_{vs} gets projected to the near clipping plane of the camera, but for depth testing, a definite z-value is needed. As z is independent from the x- and y-component, the w-component is used to find the relationship between z_{ndc} and z_{vs}:

$$z_{ndc} = \frac{z_{cs}}{w_{cs}} = -\frac{z_2 \cdot z_{vs} + z_3 \cdot w_{vs}}{z_{vs}} \tag{22}$$

Because w_{vs} equals 1 in view space, it can be dropped:

$$z_{ndc} = \frac{z_{cs}}{w_{cs}} = -\frac{z_2 \cdot z_{vs} + z_3}{z_{vs}} \tag{23}$$

Mapping (z_{vs}, z_{ndc}) to $(-n, 1)$ and $(-f, 1)$ defines the whole range of z-values in normalized device coordinates.

$$\begin{aligned} -\frac{z_2 \cdot n + z_3}{n} &= -1 \\ -\frac{z_2 \cdot f + z_3}{f} &= 1 \end{aligned} \tag{24}$$

Solving for z_2 or z_3 and inserting one into each other results in:

$$\begin{aligned} z_2 &= -\frac{f+n}{f-n} \\ z_3 &= -\frac{2\cdot f \cdot n}{f-n} \end{aligned} \tag{25}$$

So the final projection matrix looks like this:

$$M_{projection} = \begin{pmatrix} \frac{2\cdot n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2\cdot n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2\cdot f\cdot n}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{26}$$

Because the view frustum is symmetrical, where $|r| = |l|$ and $|t| = |b|$, the projection matrix can be simplified with $r + l = 0$, $t + b = 0$, $r - l = 2 \cdot r$ and $t - b = 2 \cdot t$:

$$M_{projection} = \begin{pmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & y_3 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2\cdot f\cdot n}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{27}$$

In case of screen space reflections, first, the reflected ray in view space ($\overrightarrow{R_{vs}}$) is calculated, then the vertex position P_{vs}, which is actually the reflection point P_{vs}, is added to the ray to get the reflected point R_{vs}, which can be projected using the projection matrix. Besides, the vertex position is also projected into screen space. Thus, the screen space reflected vector $\overrightarrow{R_{ss}}$ can be calculated using screen space vertex position P_{ss} and screen space reflected point R_{ss}, as shown in figure 45.The so gained screen space reflected ray can then be used for ray marching.

Another major problem is, that the z-values after the projection are not linear anymore, meaning there is a higher precision of depth at the near plane and a lower precision at the far plane. But for comparing the

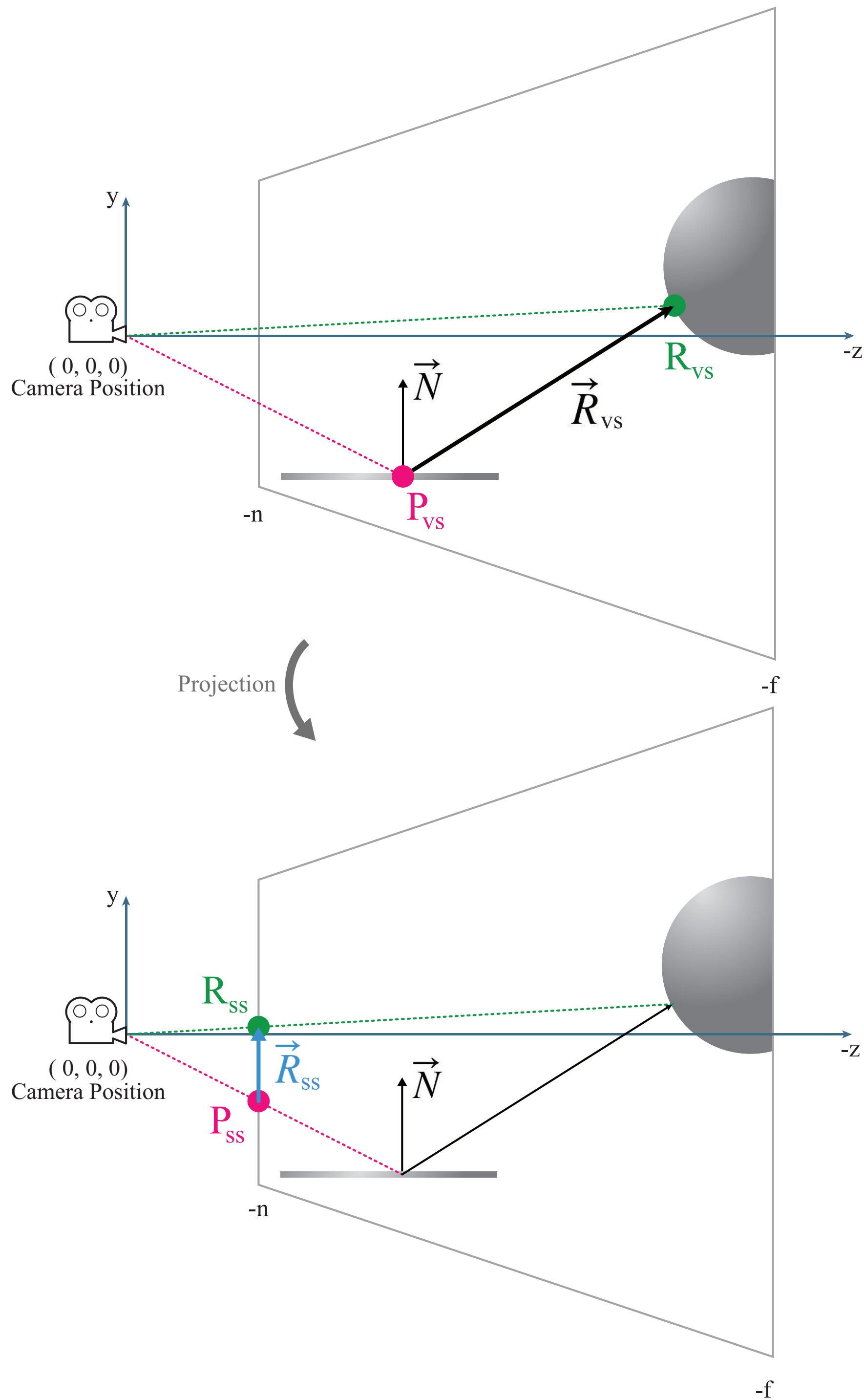


Figure. 45: Projection of the reflected ray $\vec{R}$ from view space (vs) into screen space (ss, P is the vertex position, $\vec{N}$ is the vertex normal, R is the reflected point

depths of the screen space reflection vector and the values from the depth buffer texture, a linear scale would give better results. Backward calculation of the linear depth can be used to obtain a linear depth value. The implemented shader program uses the linear depth calculation from *Geeks3D*[41]. The formula to obtain a linear depth again looks like the following:

$$z_{linear} = \frac{2 \cdot n}{f + n - z_{ndc} \cdot (f - n)} \tag{28}$$

As ray marching along the screen space reflection vector $\overrightarrow{R_{ss}}$, the texture from the depth buffer of the G-Buffer is sampled using the reflection vector, as it is shown in figure 46. Linearizing the depth values using equation 28, the depth can be compared with the z-value of the reflected ray. So the diffuse color is returned, when the z-value of the traced ray exceeds the depth value of the sampled depth texture. This can be done for every pixel on the screen, as it is done in code listings 4. This brute-force approach scales very badly, because bigger resolutions need more ray steps according to the increasing pixel count and the algorithm can get slow. A better approach is to use a similar idea to binary search, as described in code listings 5.

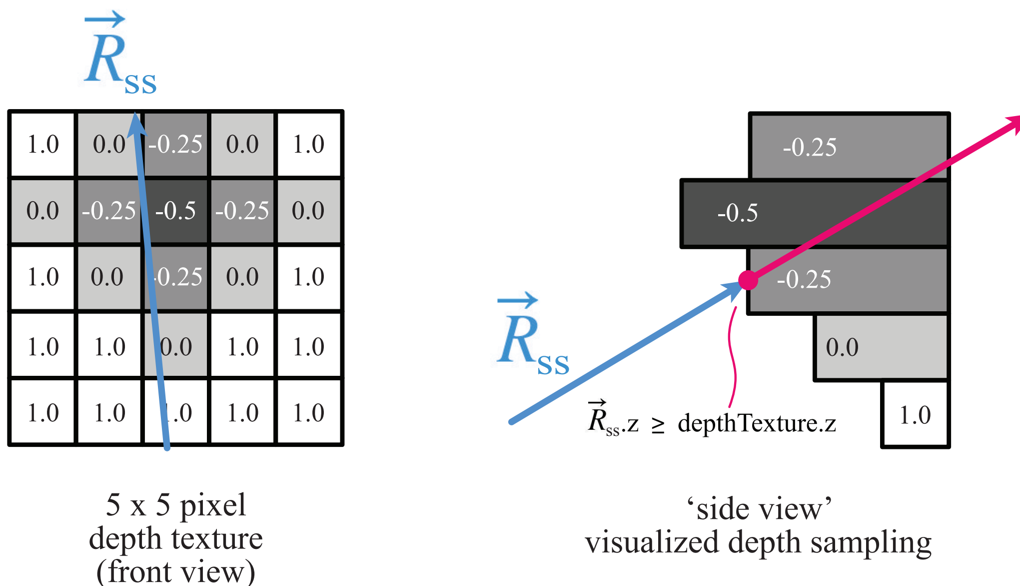


Figure. 46: Reflected ray $\overrightarrow{R_{ss}}$ is used to sample the depth texture, simplified figure

[41] www.geeks3d.com/20091216/geexlab-how-to-visualize-the-depth-buffer-in-glsl/, [accessed 02/04/2013]

Listing 4: Shader code for screen space reflections

```
//  Screen space reflections
vec4 ScreenSpaceReflections(in vec3 vsPosition, in vec3 vsNormal, in vec3 vsReflectionVector)
{
  // Variables
  vec4 reflectedColor = vec4(0.0);
  vec2 pixelsize = 1.0/vec2(Screen.Width, Screen.Height);

  // Project vertex position to screen space
  vec4 csPosition = ProjectionMatrix * vec4(vsPosition, 1.0);
  vec3 ndcsPosition = csPosition.xyz / csPosition.w;
  vec3 ssPosition = 0.5 * ndcsPosition + 0.5;

  // Project reflected vector to screen space
  vsReflectionVector += vsPosition;
  vec4 csReflectionVector = ProjectionMatrix * vec4(vsReflectionVector, 1.0);
  vec3 ndcsReflectionVector = csReflectionVector.xyz / csReflectionVector.w;
  vec3 ssReflectionVector = 0.5 * ndcsReflectionVector + 0.5;
  ssReflectionVector = normalize(ssReflectionVector - ssPosition);

  // Ray trace
  // Initial step is one pixel
  // pixelStepSize describes how much pixel at once are ray marched
  float initalStep = min(pixelsize.x, pixelsize.y);
  float pixelStepSize = 1;
  ssReflectionVector *= initalStep;

  // Sampling positions
  vec3 lastSamplePosition = ssPosition + ssReflectionVector;
  vec3 currentSamplePosition = lastSamplePosition + ssReflectionVector;

  int sampleCount = max(int(Screen.Width), int(Screen.Height));
  int count = 0;

  //****************************
  //*** Unoptimized ray trace ***
  //****************************
  while(count < sampleCount)
  {
    // Can be used to control loops
    //if(count > 150)
    //  break;

    // Out of texture space --> break
    // Because sampling from a texture, the range is from 0.0 to 1.0
    if(currentSamplePosition.x < 0.0 || currentSamplePosition.x > 1.0 ||
       currentSamplePosition.y < 0.0 || currentSamplePosition.y > 1.0 ||
       currentSamplePosition.z < 0.0 || currentSamplePosition.z > 1.0)
    {
      break;
    }

    // Update ray marching variables
    vec2 samplingPosition = currentSamplePosition.xy;
    float currentDepth = linearizeDepth(currentSamplePosition.z);
    float sampledDepth = linearizeDepth( texture(DepthTex, samplingPosition).z );

    // If currentDepth (z-value of ray) is bigger than the sampled depth from texture
```

```
    // the ray hit something
    if(currentDepth > sampledDepth)
    {
      float delta = abs(currentDepth - sampledDepth);
      if(delta <= 0.001f)
      {
        // Return the reflected color
        reflectedColor = texture(DiffuseTex, samplingPosition);
        break;
      }
    }

    // Step ray
    lastSamplePosition = currentSamplePosition;
    currentSamplePosition = lastSamplePosition + ssReflectionVector;

    count++;
  }

  // Fading to screen edges
  // Because only information in screen space is present
  // the result has some hard edges at the screen corners
  // Visual result is improved by blending out at the screen edges
  vec2 fadeToScreenEdge = vec2(1.0);
  if(fadeToEdges)
  {
    // x-direction
    fadeToScreenEdge.x = distance(lastSamplePosition.x , 1.0);
    fadeToScreenEdge.x *= distance(lastSamplePosition.x , 0.0) * 4.0;
    // y-direction
    fadeToScreenEdge.y = distance(lastSamplePosition.y , 1.0);
    fadeToScreenEdge.y *= distance(lastSamplePosition.y , 0.0) * 4.0;
  }

  return reflectedColor * fadeToScreenEdge.x * fadeToScreenEdge.y;
}
```

Listing 5: Optimized screen space reflections ray marching

```
//***************************
//*** Optimized ray trace ***
//***************************
initalStep = 1.0/Screen.Height;
pixelStepSize = user_pixelStepSize;
ssReflectionVector *= initalStep;

lastSamplePosition = ssPosition + ssReflectionVector;
currentSamplePosition = lastSamplePosition + ssReflectionVector;

int sampleCount = max(int(Screen.Width), int(Screen.Height))/10;
int count = 0;

//reflectedColor = texture2D(ReflectanceTex, vert_UV);
while(count < sampleCount)
{
  // Can be used to control loops
  //if(count > 150)
  //  break;
```

```
20
21  // Out of texture space --> break
22  // Because sampling from a texture, the range is from 0.0 to 1.0
23  if(currentSamplePosition.x < 0.0 || currentSamplePosition.x > 1.0 ||
24     currentSamplePosition.y < 0.0 || currentSamplePosition.y > 1.0 ||
25     currentSamplePosition.z < 0.0 || currentSamplePosition.z > 1.0)
26  {
27    break;
28  }
29
30  vec2 samplingPosition = currentSamplePosition.xy;
31  float sampledDepth = linearizeDepth( texture(DepthTex, samplingPosition).z );
32  float currentDepth = linearizeDepth(currentSamplePosition.z);
33
34
35  //*** Step ray ***
36  // If ray's z is bigger than sampled depth
37  // --> step backward by subtracting a tenth of the reflected vector
38  if(currentDepth > sampledDepth)
39  {
40    lastSamplePosition = currentSamplePosition;
41    currentSamplePosition = lastSamplePosition - ssReflectionVector/10.0;
42  }
43  // If ray's z is smaller than sampled depth
44  // --> step forward by adding a tenth of the reflected vector
45  else if(currentDepth < sampledDepth)
46  {
47    lastSamplePosition = currentSamplePosition;
48    currentSamplePosition = lastSamplePosition + ssReflectionVector * 10.0;
49  }
50
51  //*** Ray hit ***
52  float delta = abs(currentDepth - sampledDepth);
53  if(delta < 0.002f)
54  {
55    // Blurring dependent on viewer's distance
56    // toggleBlur is a boolean uniform
57    if(toggleBlur)
58    {
59      float f = currentDepth;
60      float blurSize = 15 * f;
61      reflectedColor = textureLod(DiffuseTex, vec2(samplingPosition.x, samplingPosition.y), 7);
62
63      int counter = 0;
64      for(float i= - blurSize/2.0; i < blurSize/2.0; i+= 1.5)
65      {
66        reflectedColor += texture2D(DiffuseTex, vec2(samplingPosition.x, samplingPosition.y + 1.0 ↩
            * i * pixelsize.y));
67        counter++;
68      }
69
70      reflectedColor /= counter;
71    }
72    // Nor blurring, just return color at sampled position
73    else
74    {
75      reflectedColor = texture2D(DiffuseTex, vec2(samplingPosition.x, samplingPosition.y));
76    }
```

```
77      }
78  
79      // Count not used here, but can be used to limit to a specific count of loops
80      count++;
81  }
```

5.4.2 Results

The SSR technique results in the visually best results of all three describes techniques, as they work directly for every reflective object in the scene, while BBR only works for the artist-defined billboards and PCCM does not provide correct reflections for object inside of the proxy geometry. Compared with a ray traced reference image, generated with ray tracer *Mental Ray*, the SSR approach yields convincing reflections, as figure 47 shows. Comparing the reflections on rounded surfaces like the body of the car, the SSR results can match up with the off-line ray traced image (while the *Mental Ray* image took about 10 minutes for rendering, the screen space technique is real-time-capable). Evaluating the pixel-per-pixel

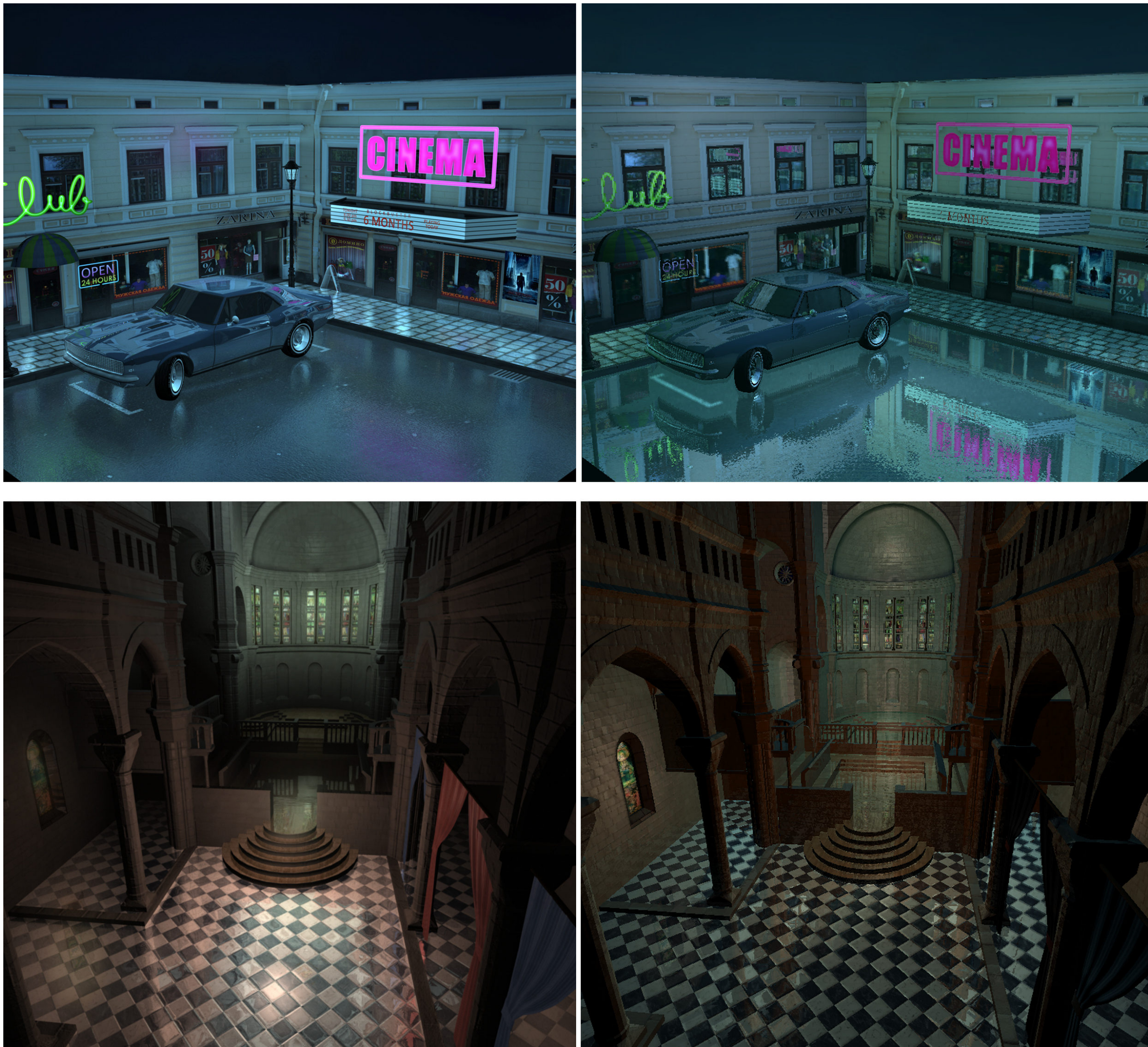


Figure. 47: Left: off-line ray traced images generated with *Mental Ray* and *Maya*, **right:** Optimized *SSR* with normal mapping, running at about $20fps$ (on a resolution of 1000 to 1000 pixels)

approach, bigger step sizes may result in ribbons and holes as shown in figure 48. Increasing the pixel-step size results in better performances, but results in inaccuracy when searching for the hit point of the ray with the scene's depth. One big disadvantage is, that SSR only works in screen space. Everything which is not seen by the viewer - in other words, is not on the screen - can not be reflected due to lack of informations at this position. To eliminate the thereby arising chamfered edges, the reflecting image can be blended out softly at the screen borders, compared in figure 49. Because of the missing texture data from the G-Buffer for all triangles, that face the viewer, also reflections can not be provided. To prevent the reflections from 'holes' that appear at these positions, a combination of SSR and the before described *Parallax-corrected cube mapping* could be used. If SSR lacks of texture information to create

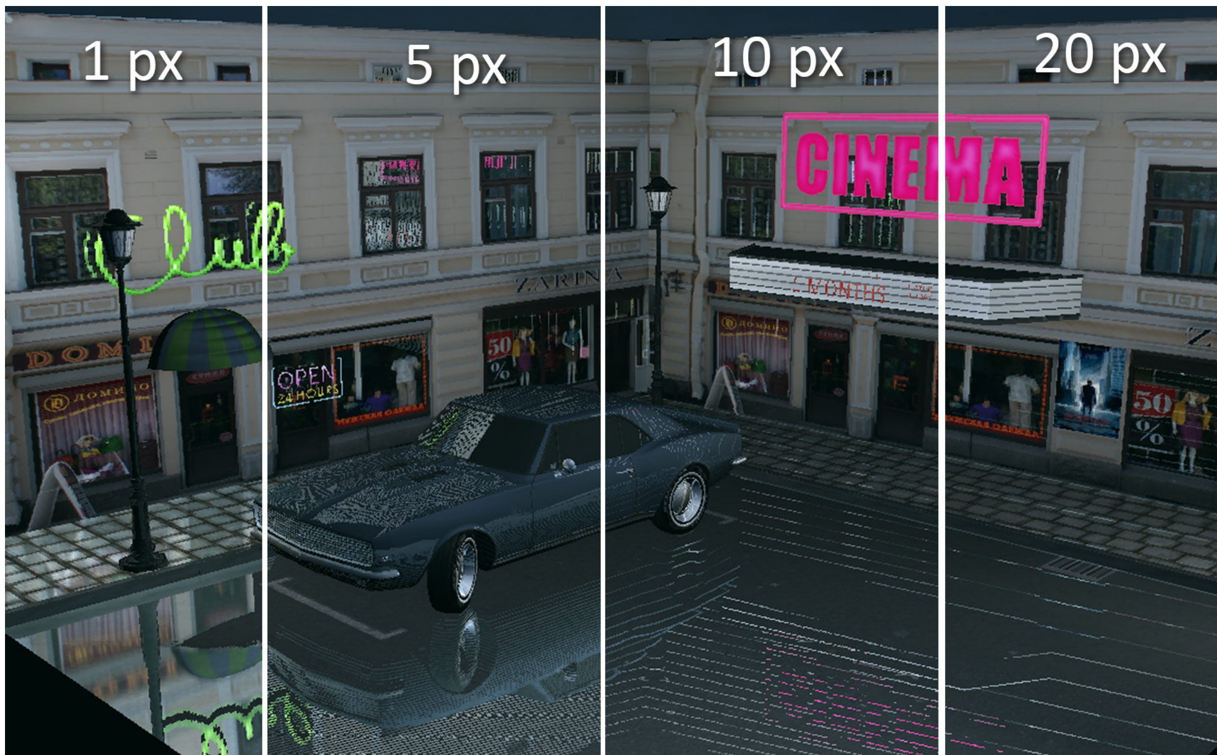


Figure. 48: Comparison of the brute-force SSR approach, ray marching with different pixel-step sizes.

reflections, the traced ray could be used to intersect with the proxy geometry and sample the cube map. This would only work for objects, that are approximated by the proxy geometry, but reflections would be present at expected positions.

Figure. 49: Optimzed SSR, **Left:** SSR creates clipped edges at the screen borders (marked with red line), due to the missing information originating from deferred shading, **right:** fading out at the screen borders creates a visually more appealing effect.

To evaluate real-time performance of SSR, frames per second have been measured for different resolutions. The framecount per second was captured for 60 seconds, while moving through the scenes. The per-pixel approach slows already at a screen resolution of 1024 to 768 pixels with average framerates of 7.29 to 8.02 for the three tested scenes. Because every pixel is ray marched, this approach scales very badly, clarified by figure 52. Better results are provided by the optimized SSR algorithm, as the charts in figure 52 illustrate. Certainly, the optimized approach even lags at a resolution of 1600 x 900 pixels and above.

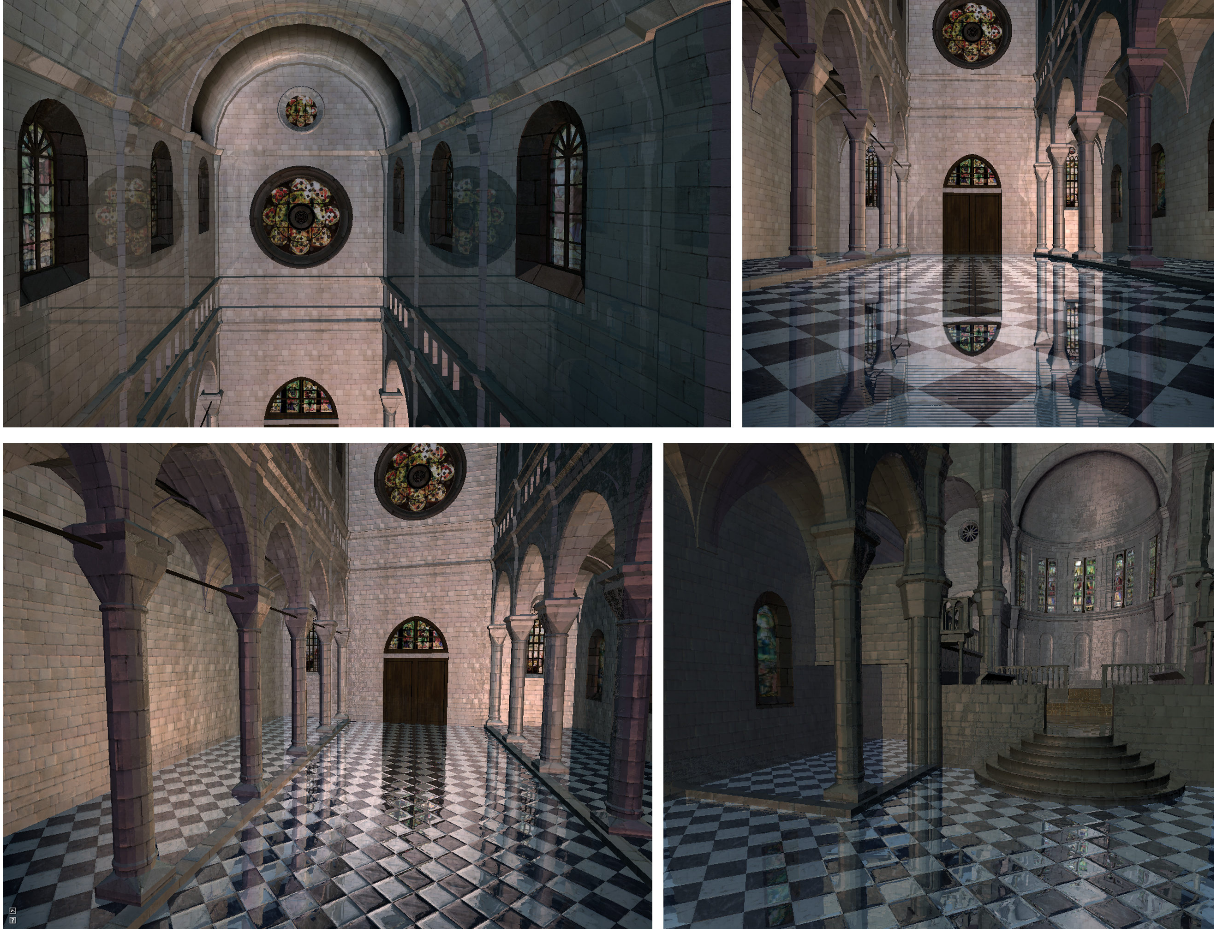


Figure. 50: Optimized SSR, tested with the cathedral scene, **top:** rendered with normal mapping, **bottom:** without normal mapping

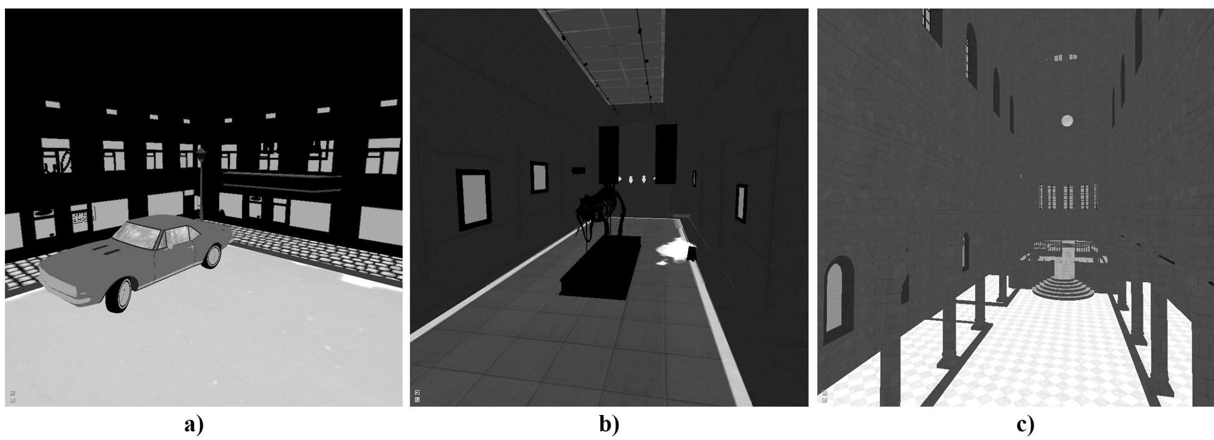
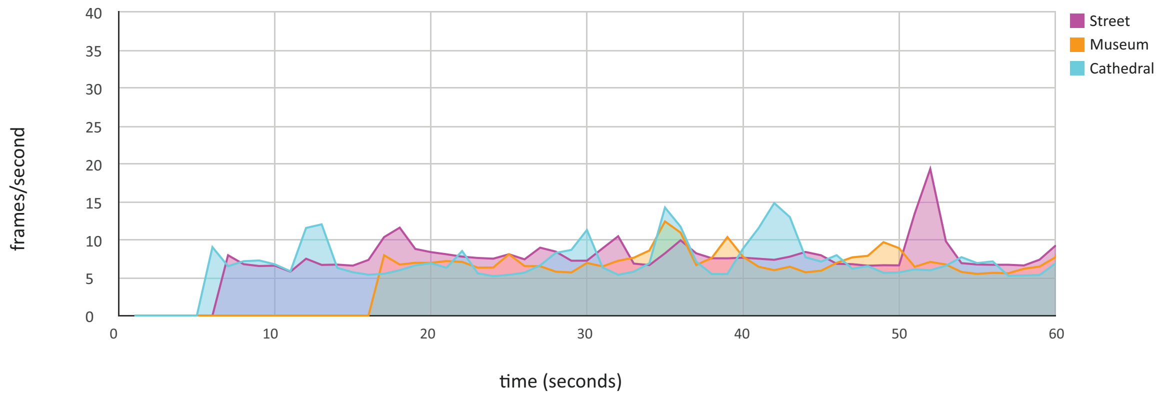
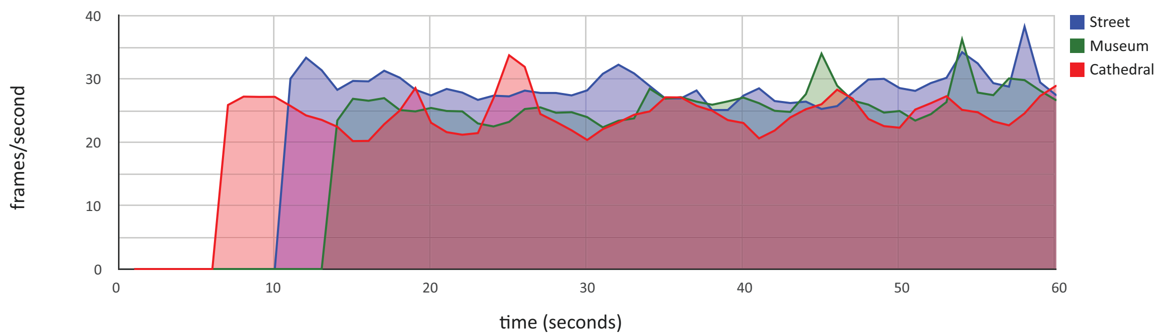


Figure. 51: The amount of reflective objects in the different scenes: **a)** street scene, **b)** museum, **c)** cathedral. The count of reflective meshes affects the performance directly

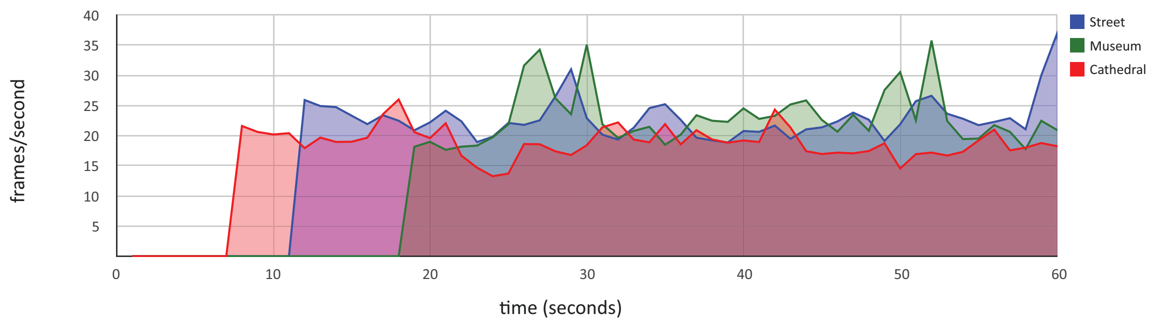
1.) SSR (unoptimzed), 1024 x 768 px



2.) SSR (optimized), 1024 x 768 px



3.) SSR (optimized), 1280 x 720 px



4.) SSR (optimized), 1600 x 900 px

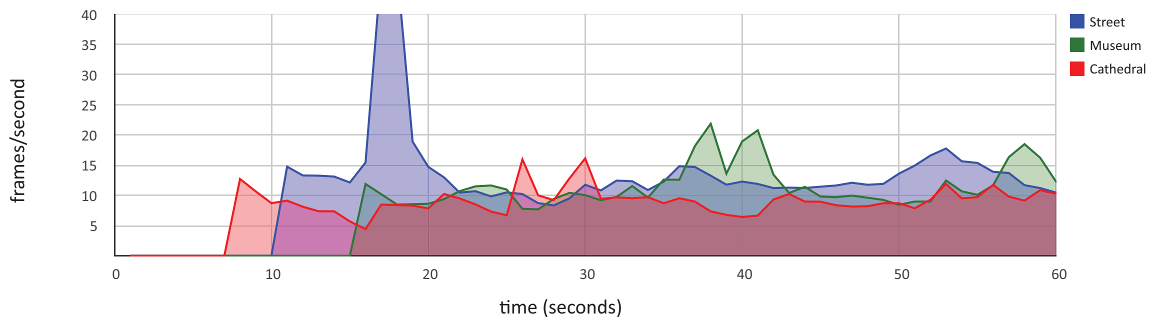


Figure. 52: Comparison of the unoptimized per-pixel SSR technique and the optimized one. Zero values at the beginning represent the loading time of the scenes. Extreme maxima are due to the amount of reflecting object on the screen (see figure 51)

6 Conclusion

Section 2 gave a physical background of light, describing important terms, such as *radiance*, to get a basic understanding of how reflections emerge and what parameters affect them. Plenty of examples from the real world and the media were given in section 2.2, showing how much reflections can visually contribute to realism and visual appeal in modern computer graphics. An overview of common approaches for rendering reflections was given, describing the most commonly used algorithms, classified in different categories. Three specific techniques have been implemented and treated in detail: Parallax-correction for cube maps, billboards for simplified high-contrast reflections and screen space reflections. They were described and evaluated in detail in section 5. All three are qualified for real-time rendering and achieve plausible visual reflections with certain drawbacks. Parallax-correction for cube maps depends on the well-known cube mapping technique and therefore can not supply correct reflections for objects inside of a room using only one environment map. Every object rather has to have its own local cube map and its approximation geometry to get correct looking reflections. The used cube maps have to be rendered offline or at run-time. Also the cube maps have to be re-generated if an object moves or when light sources change. However, the computation does not need much time and it is easy to implement, especially if environment mapping was already implemented. Parallax-corrected cube maps suit best for reflections of walls and floors in geometric rooms like corridors, halls or similar. The presented implementation could be extended with blending of different local cube maps, like it was originally described in [SZ12]. Billboards use impostor quads, that are ray-traced to generate reflections. One major problem is, that the reflections only work for billboards. Additional, the placement of the billboards has to be planed when setting up the scene. Another major problem is, that naturally, the billboard reflections lack of occlusion through other scene objects. The implementation of billboard reflections is easy and does not need to much computation time. They suit best to simulate high-contrast light reflections, as they occur in city environments at night, e.g. from neon advertisings and traffic lights. To improve the presented implementation, alpha blending could be done in a single pass to get real alpha blending, at the moment the alpha channel is just clipped at a certain value. Another big improvement that could be concerned with is the missing occlusion through other scene objects. Additionally, the billboard impostor quads could be rendered with a glow effect to give a better visual plausibility, that they are light sources. The distance dependence could also be improved by blurring the billboard textures depending on the viewer's distance to them.

Screen space reflections work only in screen space and therefore have cropped borders at the screen edges, due to missing information from the G-Buffer textures, that indeed can be blended in to get a better looking result. Therefore they afford plausible looking reflections for the whole scene and are capable for real-time renderings to a certain resolution of the screen. One major problem is the single layer depth map used for the intersection calculation of the reflected ray, that generate artifacts when the ray exits an object and there would be a second or third or even more intersections. To handle this problem, a depth-peeling approach could be used to generate a multi-layered depth map, that could be used for intersecting with the reflected ray. Also the intersection calculation could be optimized by trying different intersection-finding approaches and examining and comparing them to each other. Additionally, view dependency could be improved by blurring the sampled diffuse texture, coming from the lighting pass, dependent on the viewers distance.

All in all, this thesis gave an overview of state-of-the-art techniques for rendering reflections and describes three up to date approaches, that were examined of their visual qualities, their performance and their application in real-time rendering.

As a prospect to future real-time computer graphics, on one hand more and more ray tracing approaches will be adopted on the GPU, improving the real-time capabilities of the algorithms by better hardware and on the other hand by finding new approaches . In the near future more mixing of ray tracing and the common rasterization pipeline, like the discusses algorithms up to more complex techniques, like the shortly described voxel-cone-tracing, will be implemented until ray tracing generates decent framerates.

List of Figures

1	Perfect and imperfect reflection	3
2	Isotropic and anisotropic reflections in Unreal Engine 3	4
3	Anisotropic reflections	4
4	Urban reflections at night A	5
5	Urban reflections at night B	5
6	Urban reflections at daylight	6
7	Thin water puddles	7
8	Reflections on water	7
9	Reflections on architecture	7
10	Reflections on architecture	8
11	Reflections in art	8
12	Reflections in photography	9
13	Reflections in art	9
14	Reflections in Crysis 2	10
15	Reflections in Unreal Engine 4	10
16	Basic idea of rendering reflections	11
17	Reflections using geometry transformation	12
18	Environment mapping steps	13
19	Environment mapping example	13
20	Prefiltering environment maps	14
21	GPU ray tracing using CUDA	15
22	GPU ray tracing sampled geometry	16
23	Voxel cone tracing	17
24	UML class diagram	18
25	GUI	20
26	Render pass overview	21
27	G-Buffer	22
28	Different test scenes	23
29	Comparison of cube mapping and PCCM	24
30	AABB in Maya	25
31	Parallax-corrected cube mapping	25
32	PCCM results (street scene)	27
33	PCCM results (cathedral)	27
34	BBR example (Unreal Engine 3)	28
35	Billboard placement in Maya	28
36	Billboard placement in a scene	29
37	Standard triangle for billboard-intersection	30
38	Ray-triangle transformation	30
39	BBR results (street scene)	34
40	BBR results (museum scene)	34
41	BBR results (missing occlusion)	35
42	BBR framerate comparison	35
43	View frustum	37
44	Clip space to normalized device coordinates	37
45	Projection of the reflected ray	39
46	Sampling depth texture	40
47	Comparison of SSR to off-line ray traced image	45
48	SSR comparison of different pixel step sizes	46
49	SSR results (fade-out at screen borders)	46
50	SSR results (cathedral)	47

51 Reflectivity of different scenes 47
52 Framerate comparison of optimized and per-pixel SSR 48

Listings

1 Shader code for Parallax-corrected cube mapping 26
2 Shader code for billboard reflections 32
3 Shader code for ray-billboard intersection 32
4 Shader code for screen space reflections 41
5 Optimized screen space reflections ray marching 42

List of Algorithms

1 Basic ray tracing algorithm 14
2 Pseudo code for parallax-corrected cube mapping 24
3 Billboard reflections algorithm 29
4 Pseudo code of the screen space reflections algorithm 36

References

[Ahn11] Song Ho Ahn. OpenGL Tutorials. http://www.songho.ca/opengl/index.html, [accessed 02/04/2013], 2005 - 2011.

[Ble09] Neil Blevins. Anisotropic Reflections. http://www.neilblevins.com/cg_education/aniso_ref/aniso_ref.htm, 2009.

[BN76] James F. Blinn and Martin E. Newell. Texture and Reflection in Computer Generated Images. *Commun. ACM*, 19(10):542--547, October 1976.

[CHCH06] Nathan A. Carr, Jared Hoberock, Keenan Crane, and John C. Hart. Fast GPU ray tracing of dynamic meshes using geometry images. In *Proceedings of Graphics Interface 2006*, GI '06, pages 203--209, Toronto, Ont., Canada, Canada, 2006. Canadian Information Processing Society.

[CNS+11a] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing: A preview. Poster ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D). Best poster award., feb 2011.

[CNS+11b] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive Indirect Illumination Using Voxel Cone Tracing: An Insight. Siggraph 2011 Talk, aug 2011.

[Hen09] Justin Hensley. Shiny, Blurry Things. developer.amd.com/wordpress/media/2012/10/09-hensley-BPS.pdf, [accessed 03/04/2013], 2009.

[IL12] Nop Jiarathanakul Ian Lilley, Sean Lilley. Real-Time Voxel Cone-Tracing. cis565-fall-2012.github.com/lectures/11-01-GigaVoxels-And-Sparse-Textures.pdf, [accessed 04/04/2013], 2012.

[KVHS00] Jan Kautz, Pere-Pau Vázquez, Wolfgang Heidrich, and Hans-Peter Seidel. A Unified Approach to Prefiltered Environment Maps. http://dl.acm.org/citation.cfm?id=647652.732274, [accessed 04/04/2013], 2000.

[Lag12] Sébastien Lagarde. Water Drop 1 – Observe Rainy World. http://seblagarde.wordpress.com/2012/12/10/observe-rainy-world/, 2012. [Online, accessed 19/03/2013].

[Lil12] Ian Lilley. Real-Time Screen-Space Reflections in OpenGL. http://de.scribd.com/doc/91201173/Paper-IanLilley, 2012.

[MD11] Martin Mittring and Bryan Dudash. The Technology Behind the DirectX 11 Unreal Engine "Samaritan" Demo. http://udn.epicgames.com/Three/rsrc/Three/DirectX11Rendering/MartinM_GDC11_DX11_presentation.pdf, 2011. [Online, accessed 17-March-2013].

[Mit12] Martin Mittring. The Technology Behind the “Unreal Engine 4 Elemental Demo”. www.unrealengine.com/files/misc/The_Technology_Behind_the_Elemental_Demo_16x9_(2).pdf, [accessed 04/04/2013], 2012.

[MPS05] Chunhui Mei, Voicu Popescu, and Elisha Sacks. The Occlusion Camera. *Computer Graphics Forum*, 24:335--342, 2005.

[MT97] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools*, 2(1):21--28, October 1997.

[NK11] Tiago Sousa Nickolay Kasyan, Nicolas Schulz. Secrets of CryENGINE 3 Graphics Technology. In *ACM SIGGRAPH 2011 Courses*, SIGGRAPH 2011, 2011.

[OR98] Eyal Ofek and Ari Rappoport. Interactive reflections on curved objects. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 333--342, New York, NY, USA, 1998. ACM.

[PBMH02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 703--712, New York, NY, USA, 2002. ACM.

[PHR+09] Voicu Popescu, Kyle Hayward, Paul Rosen, Chris Wyman, Voicu Popescu, Kyle Hayward, Paul Rosen, and Chris Wyman. Non-pinhole impostors. http://www.cs.purdue.edu/research/technical_reports/2009/TR%2009-006.pdf, 2009. [accessed 01/03/2013].

[PMDS06] Voicu Popescu, Chunhui Mei, Jordan Dauble, and Elisha Sacks. Reflected-Scene Impostors for Realistic Reflections at Interactive Rates. *Comput. Graph. Forum*, 25(3):313--322, 2006.

[Ral07] Kristóf Ralovich. Implementing and Analyzing A GPU Ray Tracer. http://www.cescg.org/CESCG-2007/papers/TUBudapest-Ralovich-Kristof.pdf, [accessed 04/04/2013], 2007.

[SAG+05] Peter Shirley, Michael Ashikhmin, Michael Gleicher, Stephen Marschner, Erik Reinhard, Kelvin Sung, William Thompson, and Peter Willemsen. *Fundamentals of Computer Graphics, Second Ed.* A. K. Peters, Ltd., Natick, MA, USA, 2005.

[Sha10] Christopher Shane. Integration of Ray-Tracing Methods into the Rasterisation Process. www.scss.tcd.ie/postgraduate/msciet/current/Dissertations/0910/Christopher.pdf, [accessed 04/04/2013], 2010.

[SKALP05] László Szirmay-Kalos, Barnabás Aszódi, István Lazányi, and Mátyás Premecz. Approximate Ray-Tracing on the GPU with Distance Impostors. *Comput. Graph. Forum*, 24(3):695--704, 2005.

[SKUP+09] László Szirmay-Kalos, Tamás Umenhoffer, Gustavo Patow, László Szécsi, and Mateu Sbert. Specular Effects on the GPU: State of the Art. *Computer Graphics Forum*, 28(6):1586--1617, 2009.

[SZ12] Lagarde Sébastien and Antoine Zanuttini. Local image-based lighting with parallax-corrected cubemaps. In *ACM SIGGRAPH 2012 Talks*, SIGGRAPH '12, pages 36:1--36:1, New York, NY, USA, 2012. ACM.

[Tat06] Natalya Tatarchuk. Artist-directable real-time rain rendering in city environments. In *ACM SIGGRAPH 2006 Courses*, SIGGRAPH '06, pages 23--64, New York, NY, USA, 2006. ACM.

[Wil] WilliamDiSanto. Advanced ComputerGraphics CS 563: Screen Space GI Techniques: Real-Time. http://web.cs.wpi.edu/~emmanuel/courses/cs563/S12/slides/cs563_Will_DiSanto_ssdo_wk6_p1.pdf, [accessed 01/04/2013].

[YWY08] Xuan Yu, Rui Wang, and Jingyi Yu. Interactive Glossy Reflections using GPU-based Ray Tracing with Adaptive LOD. *Comput. Graph. Forum*, 27(7):1987--1996, 2008.