FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Interactive Curved Reflections in Large Point Clouds

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computergraphik/Digitale Bildverarbeitung

eingereicht von

Reinhold Preiner
Matrikelnummer 0430380

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer/in: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

Wien, 19.05.2010

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Erklärung zur Verfassung der Arbeit

Reinhold Preiner
Murlingengasse 56/4
1120 Wien

"Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen – die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe."

Ort, Datum:

Unterschrift:

Abstract

In the field of real-time rendering, computer graphics observes a continuously growing power of visualizing of scenes of continuously growing complexity. In the past few years, a number of rendering techniques have been developed that let the quality of the rendered images converge towards photorealism. Especially in the field of realistic scene illumination, Global Illumination (GI) techniques represent an important field of research. Previous to this work, for the first time we have applied a GI algorithm also to point clouds, which enables us to achieve realistic illumination of diffuse and glossy objects in real-time.

This thesis elevates the power of visualization of this GI-Renderer to the next level. For the first time, it implements a realistic, physically based rendering of mirroring surfaces also for point clouds.

Current real-time approaches addressing curved mirroring surfaces in polygon scenes either are extremely imprecise or cannot handle each arbitrary type of surface. Especially concave surfaces represent a significant difficulty for current physically based methods. Up to now, physically correct mirror reflections on complex surfaces can only be produced by offline algorithms.

We introduce a novel rendering technique called *Screen-space curved reflections*, which enables us to produce physically correct mirror reflections on arbitrarily complex surfaces. Our method bases on the approach, for each point in the scene to find the pixel in the framebuffer that contains its reflecting surface point. This is achieved by the application of a fast 2D root finding in a new error function called *mirror-space error function*. Although our method raises high demands on the hardware, we are able to render common scenes at interactive frame rates.

Kurzfassung

Im Bereich des Echtzeitrenderings verzeichnet die Computergraphik eine immer rascher zunehmende Darstellungsfähigkeit für immer größere und komplexere Szenen. In den letzten Jahren haben sich eine Vielzahl an Rendering-Techniken etabliert, die die Qualität der generierten Bilder in konventionellen Polygonszenen immer mehr an den Fotorealismus konvergieren lässt. Vor allem im Teilbereich der realistischen Beleuchtung von Szenen stellen Global-Illumination (GI) Techniken aktuell ein wichtiges Forschungsgebiet dar. Im Vorfeld dieser Arbeit haben wir erstmals einen GI-Algorithmus auf riesige Punktwolkenszenen angewandt, mit dem wir eine realistische Beleuchtung von diffusen und glänzenden Objekten in Echtzeit erzeugen können.

Die vorliegende Diplomarbeit setzt die Leistungsfähigkeit dieses GI-Renderers auf die nächste Ebene. Sie implementiert erstmals die Darstellung von Oberflächenspiegelungen auch für Punktwolkenszenen.

Bisherige Echtzeit-Rendering-Techniken für gekrümmte Spiegelungen in Polygonszenen sind entweder extrem ungenau, oder unterstützen nicht jede beliebige Art von Oberfläche. Speziell konkav gekrümmte Oberflächen stellen bei aktuellen physikalisch korrekten Ansätzen ein Problem dar. Bisher können korrekte Spiegelungen auf komplexeren Oberflächen nur durch Offline-Algorithmen erzeugt werden.

Wir haben eine neuartige Technik namens *Screen-Space Curved Reflections* entwickelt, die es erlaubt, physikalisch korrekte Spiegelungen auf beliebig komplexen Oberflächen darzustellen. Die Methode basiert auf dem Ansatz, für jeden Punkt in der Szene jenen Pixel im Framebuffer zu suchen, der dessen reflektierenden Oberflächenpunkt enthält. Wir erreichen dies durch eine effiziente Nullstellensuche in einer von uns eingeführten Fehlerfunktion, der *mirror-space error function*. Obwohl unsere Methode hohe Anforderungen an die Hardware stellt, sind wir in der Lage, gängige Szenen bei interaktiven Bildwiederholungsraten darzustellen.

Acknowledgements

I'd like to thank my supervisor Michael Wimmer for his support, the helpful input and the creative discussions. He motivated me to follow my idea about SSCR and allowed me to develop it in terms of this thesis.

Thanks to the people from the Institute of Computer Graphics at the TU Vienna, for the wonderful working atmosphere and for letting me feel welcome in the institute so fast.

I'd also like to thank Andrea Weidlich for providing me her expertise and contributing images for this thesis.

Further thanks to Stefan Reinalter for providing his knowledge and experience in organizational issues.

Thanks to the awesome guys from the CGC, who stage the best console evenings ever, and also showed understanding for my rare availability during the time of creating this thesis.

Many thanks to my family, which supported me over all the years of my study. Thanks to my grandparents for their nut kernels and mineral water supplies.

Most gratitude however I owe to my beloved Marge, without whom I would not have been able to do any of this. She showed incredible understanding and patience in helping, supporting and motivating me during the past few weeks.

Contents

Chapter 1

Introduction

In image synthesis, there are several factors by which the quality of a rendered image is measured. First of all, they depend on the demands stated for the renderer. An overall believable visual impression can as well be an important target as a clean and artifact-free per-pixel-correctly rendered image. The work in this thesis bases on a renderer that deals with low-quality geometry: point clouds. Here we face a shift in the demands in relation to polygon scene renderers. We can hardly claim artifact-free per-pixel-correctly rendered images, especially when dealing with point splats that are often required to obtain a dense image of a point cloud object. Thus, in point clouds we rather pursue to produce generally believable images so far, always trying to enhance realism in our rendered scenes.

An important aspect of the realistic appearance of objects in computer graphics as well as in everyday life, are mirror reflections. From the concave surface reflections on our breakfast coffee spoon over the metal door knobs on our doors at home, up to the warped projections of exterior buildings and trees on the curved shape of our car, complex mirror reflections are present in a wide range of everyday scenes.

In this thesis, we introduce a new method to simulate curved reflections in large point cloud scenes at interactive frame rates.

The first chapter provides an overview of the scope of the thesis and introduces our point rendering framework *Terapoints* and the global illumination renderer this work bases on. Finally, it points out our main contributions and gives an overview over the structure of this thesis.

1.1 Scope of the work

In this thesis, we introduce a novel technique called *Screen-Space Curved Reflections* (SSCR) for applying mirror reflections to an interactive global illumination renderer for point clouds [Pre10]. Although this thesis focuses on the implementation of one mirror bounce, the architecture of the method

is designed to be easily extended to support even multiple mirror bounces by iterative application of the technique.

We provide an introduction to the idea behind the approach, the methods and algorithms used, and how it is embedded into the global illumination rendering system applied to large point cloud scenes. Further, we describe the implementation of our algorithm in detail and analyze the difficulties of the method and how they are addressed by our implementation.

1.2 The Terapoints point renderer

The framework for the implementation of this thesis is given by the *Terapoints*[1] point renderer developed at the Institute for Computer Graphics and Algorithms at the Vienna University of Technology. This application is specially designed for instant visualization of huge point clouds, mainly applied for in-situ visualization of point data gathered from range scanners.

The implementation of this technique is based upon and embedded into a global illumination rendering system for such point clouds [Pre10].

1.3 Main contributions

The following list points out the main contributions of this thesis.

- We introduce a novel rendering technique called *screen-space curved reflections* (SSCR) that allows for interactive, correctly warped environment reflections on arbitrarily curved surfaces, without the need for many-viewpoint approaches, surface tesselations or scene subdivisions. Basically, the method tries to find the one pixel within a mirroring region in the frame-buffer, that best reflects a given scene point into the viewpoint. We achieve this by applying a fast screen-space root finding algorithm in this region, minimizing the deviation of the current reflection vector to the eye vector of a mirroring surface point.

- We present a new function over a reflective surface, the *mirror-space error function*, that determines the error of a scene point's reflection vector to the eye-vector in a new affine space called *mirror space*.

- We show an efficient way to perform 2D root finding in screen space by approximating the 2D problem by two individual 1D algorithms, that can be compared to bracketing and bisection. Since the method is just

[1] Formerly known as *Scanopy*: www.cg.tuwien.ac.at/research/projects/Scanopy

approximative, there are cases where our method is not guaranteeing to actually find an intended root. However, for our task, it still provides sufficient robustness to produce dense mirror reflections in most cases.

- We provide a new mechanism that enhances former G-Buffer based global illumination techniques in a way that allows for fast global illumination shading even for multiple-bounced mirrored scene parts of the image. This is achieved by iterative substitution of mirroring G-Buffer elements.

1.4 Structure of this thesis

This thesis is structured into the following chapters:

- Chapter 2 gives an overview over the background of this thesis. First it explains the basics of BRDFs (*Bidirectional Reflectance Distribution Function*) by means of Kajiya's Rendering Equation [Kaj86], especially focusing on mirror reflections. Further, it evaluates related work and current state of the art in the field of curved surface mirror reflections, analyzing the strengths and weaknesses of those methods. Finally, it describes our current method for global illumination rendering in point clouds [Pre10], and shows how the work of this thesis can enhance its power of visual simulation.

- Chapter 3 introduces our new approach, points out its prerequisites and describes our technique for rendering screen space curved reflections in point clouds in detail. Additionally, we analyze the limits and difficulties of our technique and show how it was improved in order to address those issues.

- Chapter 4 focuses on the details of our implementation, and describes how the method can be embedded into a given G-Buffer based GI rendering pipeline by exploiting already given intermediate data.

- Chapter 5 shows the results of our implementation. It gives a detailed evaluation of the method by analyzing the dependence of image quality and performance on the scene complexity and the parameters of the algorithm.

- Chapter 6 summarizes the contents of the thesis and its contributions. It shortly describes a possible application of our method to polygon scenes, and finally discusses other future work.

Chapter 2

Background

2.1 The Rendering Equation

One of the most common and challenging tasks in computer graphics is the realistic illumination of virtual scenes. In general, it is the attempt of simulating and predicting the behavior of light as it propagates through the scene, originating at the light source and finally reaching the observer's viewpoint, being influenced by the interaction with objects' surfaces and transparent media on its way.

An idealized and complete description of the process of light propagation in a scene is given by the *Rendering Equation*, which was introduced by James Kajiya in 1986 [Kaj86] . Equation 2.1 shows the *hemispherical form*, a common notation of the rendering equation.

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_\Omega \rho(p, \omega_i, \omega_o) L_i(H(p, -\omega_i), \omega_i) \cos\theta_i d\omega_i \qquad (2.1)$$

Basically, the rendering equation formulates the relation between the light outgoing from a given surface point p and the light incoming from the whole hemisphere over p with respect to the characteristics of the surface at p. Figure 2.1 illustrates this concept.

Elements of the equation In the above formulation, the rendering equation is a function of the surface point p and the direction of outgoing light ω_o. The sum of energy radiating from p into direction ω_o is described by the sum of a) energy emitted by the material and b) energy reflected at p into that direction. Since the latter means an evaluation of an infinite number of possible incoming light directions, the reflective part of the equation is integrated over the whole hemisphere Ω above p.

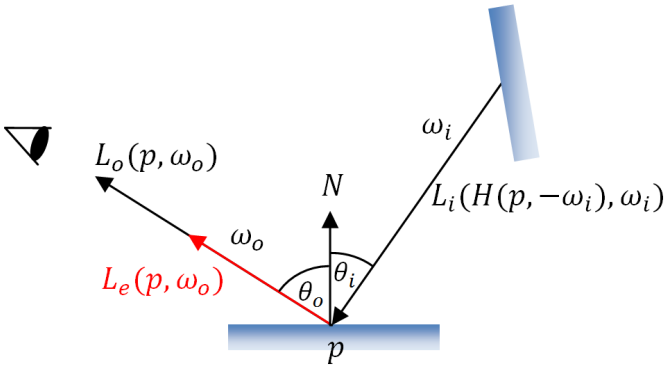


Fig. 2.1: Illustration of the light propagation concept as formulated by the rendering equation.

$L_e(p, \omega_o)$	Energy emitting from the surface point p into the outgoing direction ω_o.
$\rho(p, \omega_i, \omega_o)$	Function of the surface point p and the incoming and outgoing light direction ω_i) and ω_o, that evaluates the relative amount of light incident from ω_i) that is reflected into ω_i. This function is called *Bidirectional Reflectance Distribution Function* (BRDF) and is discussed in more detail in 2.2.
$L_i(H(p, -\omega_i), \omega_i)$	Amount of energy incident at p from direction ω_i. The inner function H is a *visibility function* of p and ω_i, which takes a possible occlusion of p from the incoming direction into account.
$\cos\theta_i$	This factor correctly attenuates the irradiance at p given by L_i, depending on the angle θ of the incoming light direction. The density of illumination per unit projected area falls off by the cosine of θ, resulting in zero irradiance at completely flat incoming light. Figure 2.2 illustrates this fact.

Evaluation of the rendering equation In computer graphics, simulating light propagation based upon this rendering equation faces two major problems.

First, the function L_o is recursive, i.e. the incoming energy L_i at p equals the outgoing energy L_o at some previous surface point p', which in turn has

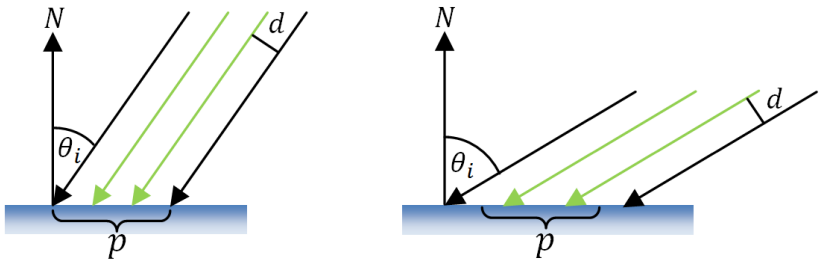


Fig. 2.2: Parallel rays hitting a surface. r represents the intensity of incident light. With decreasing angle θ, the energy incident on a given area around p is attenuated.

to be evaluated by L_o at p'' and so on, meaning that the correct evaluation of the illumination of p from a single direction would require following back the incoming light's way over all of its previous surface reflections, fast diverging in calculation complexity.

Second, a correct evaluation of the integrand over Ω is rarely possible due to the infinity of possible incoming light directions within the hemisphere over p. Thus this integrand can at best be approximated by convenient sampling of Ω. A broad discussion on these issues is given by [DBB02] and [SK00].

2.2 Bidirectional Reflectance Distribution Functions

Evaluating the rendering equation, the surface properties participating in the continuous reflection process of light significantly influence both the properties the light and its amount of energy, thus affecting the visual appearance of a scene. A comprehensive model for describing the properties of a surface is given by the *Bidirectional Reflectance Distribution Function* (BRDF).

The BRDF describes the relative amount of light of a given wavelength ϕ incident at some surface point p from an incoming direction ω_i) that is reflected into an outgoing direction ω_o) (see Figure 2.3). In its simplest form, it is a function of 6 degrees of freedom, with the following parameters:

- location of the reflecting surface point p (1 DOF)
- incoming and outgoing light direction ω_i and ω_o, each given by two angles in polar coordinates (4 DOF)
- wavelength of the reflected light (1 DOF)

The latter has to be considered since for different materials, different wavelengths are reflected in a different way, i.e in different directions and

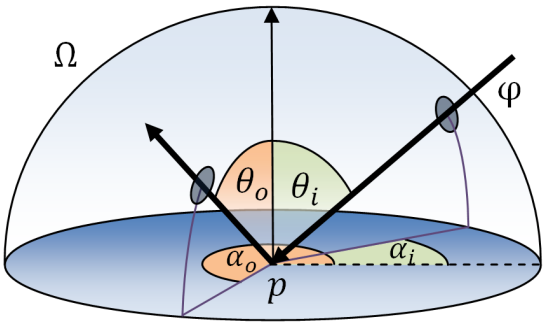


Fig. 2.3: Schematic illustration of a BRDF, evaluating the amount of energy reflected into a given outgoing direction from a given incoming direction, for a given wavelength ϕ.

in different intensity. More complex forms of the BRDF can also take light polarisation, fluorescence or phosphorescence into account, leading to up to 12 degrees of freedom. When modeling the behavior of light interacting with transparent objects, we have to extend to *Bidirectional Scattering Distribution Functions* (BSDFs) respectively to *Bidirectional Subsurface Scattering Distribution Functions* (BSSDF), which fully describe both reflection and transmission of an incident light ray [Vea98].

Characteristics of a BRDF As the name suggests, BRDFs are *bidirectional*, meaning that for two given directions ω_i and ω_o, it evaluates the same reflectance for both forward and backward light propagations, i.e. $\rho(p, \omega_i, \omega_o) = \rho(p, \omega_o, \omega_i)$. This characteristic is called *Helmholtz Reciprocity* [Nic65].

Another important law for BRDFs is *energy conservation*, stating that the amount of reflected energy is always less or equal to the amount of incident energy, i.e. $\rho(p, \omega_i, \omega_o) \leq 1$.

A simplified BRDF model In real-time computer graphics, an exact representation of this BRDF is often neglected in favor of higher performance. A widely used simplification model is the approximation of a BRDF by the weighted combination of some idealized extrema. We can roughly differ between the following main categories of the BRDF:

- **Perfectly diffuse surfaces:** Diffuse surfaces are perfect scatterers, distributing incoming light equally to all directions above the reflective point's hemisphere. Their behavior is prominently described by the *Lambertian Law*, that sets the intensity of the reflected light in direct

proportion to the cosine of the incident angle (see Figure 2.4). Diffuse objects lack any gloss or cues about the environment of the object.

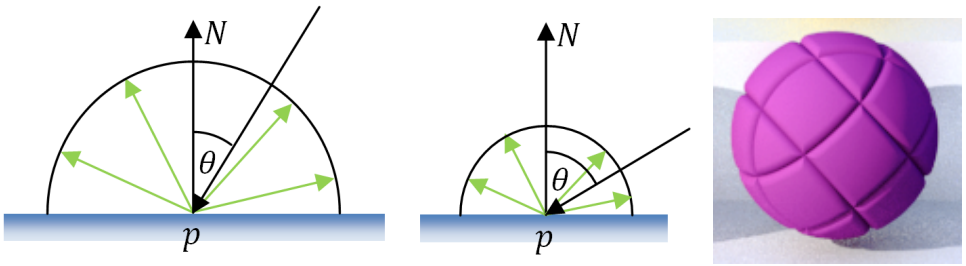
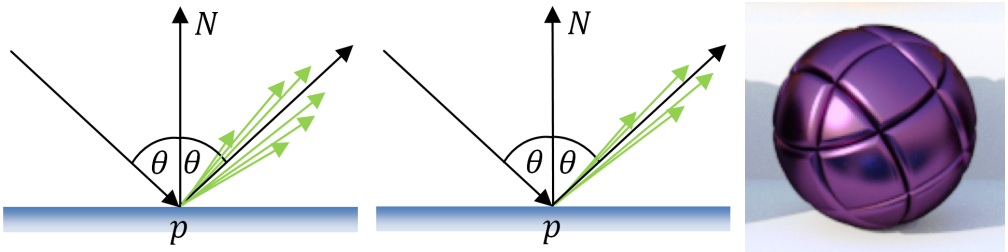


Fig. 2.4: Left: Diffuse reflection and Lambert's cosine law. Light is scattered equally to each direction from the surface. Its intensity falls off as the angle of incident light increases, which is expressed by the cosine of θ. Right: Lambertian surface (Image courtesy of Andrea Weidlich).

- **Glossy surfaces:** These surfaces scatter part of the incoming energy roughly into the direction of the perfect reflection vector, producing glossy highlights on the surface. This behavior is heuristically described by e.g. the Phong [Pho75] or the Blinn [Bli77] reflection model, which adds a specular term to the diffuse reflection model of a surface. The variance of the specularly reflected directions around the perfect reflection vector determines the degree of glossiness. The lower the variance, the glossier a surface appears. With vanishing variance, all light rays are reflected in direction of the perfect reflection vector, converging the perfect mirror behavior. The amount of glossiness is normally described by an exponent of the cosine of the angle between the actual reflection direction and the eye direction. Figure 2.5 illustrates the specular reflection model.

The method of representing each possible BRDF by a combination of these reflectance types is only a rough approximation. A physically more plausible BRDF description is given by the Cook-Torrance model [CT81], consists of a a perfectly diffuse Lambertian and a specular part which is represented by a microfacet structure. These microfacets are considered as perfectly specular reflectors that are randomly oriented according to a Gaussian distribution function and control the roughness of the surface. Reflectance attenuation is solely given by the self-shadowing or masking effects that appear on microscopic level, described by a geometry term.

In the following we will focus on an extremum of specular reflective surfaces: perfect mirrors. We will discuss their BRDF and the implications they have for the rendering equation.



(a) low Phong exponent (b) high Phong exponent

Fig. 2.5: Left: Specular reflection described by the Phong model. Incoming light is scattered in direction around the mirror reflection vector. With growing exponent, the variance decreases and the surface looks glossier. Right: Pure Phong surface (Image courtesy of Andrea Weidlich).

2.3 Mirror reflections

2.3.1 Simplifying the BRDF

Perfect mirrors represent a special borderline case for a BRDF. Each incoming light ray is reflected exactly by the surface normal based on the simple reflection function:

$$\omega_o(\alpha_o, \theta_o) = R(\omega_i(\alpha_i, \theta_i)) = \begin{cases} \alpha_o = \pi + \alpha_i \\ \theta_o = \theta_i \end{cases} \tag{2.2}$$

Thus, for each outgoing direction ω_o at a given surface point p, there is only exactly one incoming direction ω_i contributing energy, while for the rest of the hemisphere Ω over p, the BRDF evaluates to zero. In this case, the BRDF can be formulated using a *Dirac-delta* function [SK00]:

$$\rho(p, \omega_i, \omega_o) = k_s \frac{\delta(\omega_o^R - \omega_i)}{\cos\theta_o} \tag{2.3}$$

where k_s is an energy attenuation factor, ω_o^R is the mirror reflected direction of ω_o, and θ_o is the outgoing angle. Inserting this mirror BRDF into the rendering equation 2.2, leads to the following simplification of the integral over Ω:

$$\begin{aligned}
&\int_\Omega \rho(p,\omega_i,\omega_o) L_i(H(p,-\omega_i),\omega_i)\cos\theta_i d\omega_i \\
&= \int_\Omega k_s \frac{\delta(\omega_o^R-\omega_i)}{\cos\theta_o} L_i(H(p,-\omega_i),\omega_i)\cos\theta_i d\omega_i \\
&= k_s \frac{1}{\cos\theta_o} L_i(H(p,-\omega_o^R),\omega_o^R)\cos\theta_i \\
&= k_s L_i(H(p,-\omega_o^R),\omega_o^R)
\end{aligned} \tag{2.4}$$

Note that the original attenuation term $\cos\theta_i$ is eliminated at the incident direction ω_o^R due to the equal incoming and outgoing angles θ_i and θ_o.

If we assume mirror surfaces to be wavelength-conserving (meaning they do not change the color of the reflected light), the evaluation of the rendering equation is reduced to a multiplication of the incoming light with a weight factor k_s, determining which percentage of the incoming light is mirror reflected. For perfect mirrors, without any subsurface characteristics, this factor is 1, and – neglecting emission – the complete rendering equation can be written as

$$L_o(p,\omega_o) = L_i(H(p,-\omega_o^R),\omega_o^R) \tag{2.5}$$

pointing out that the outgoing light from p in direction ω_o simply equals the incoming light at p from the mirrored direction ω_o^R, under additional consideration of a possible occlusion of p from ω_o^R. We see that for perfect mirrors the difficulties raised by the complexity of the original rendering equation are reduced by eliminating the integrand over Ω, and with it the problem of infinite directions ω_i and approximation by sampling. Evaluating mirror reflections, we have to handle two remaining problems:

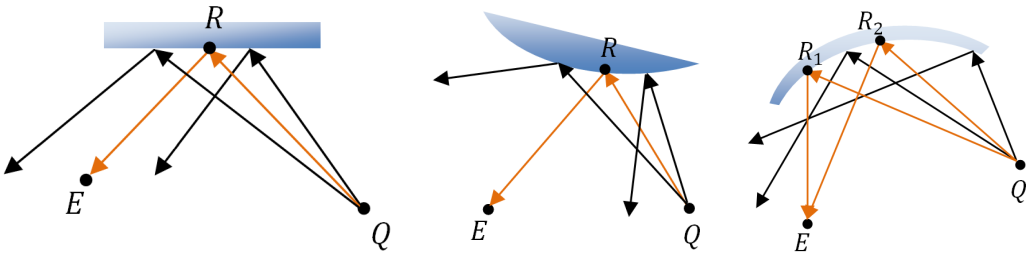
- **Visibility:** Computing the outgoing light from p in direction ω_o requires visibility information to be taken into account. In other words, the object that is observed in an mirroring surface at p from direction ω_o is the object, visible from p in direction ω_o^R.

- **Recurrence:** In the form of equation (2.5), the rendering equation is still self-containing, meaning that the evaluation of light propagation along a given path still requires backtracking of light rays in an iterative way.

2.3.2 Mirror reflections on different surface shapes

The properties of a mirroring BRDF reduce the evaluating of a surface's reflection behavior to a much easier task than for general BRDFs. Surveying a mirroring reflection process globally for a whole surface area, we can point out several observations, depending on the shape of the mirror surface.

For planar and convex mirror surfaces, we make the following observation: Given a scene point Q and view point E, we can find at most one point on the surface, that reflects Q into the direction of E. Thus, we have an one-to-one function mapping of a visible point Q into the viewport window of the camera. For concave surfaces, multiple mappings are possible, leading to an one-to-n relation. Figure 2.6 illustrates this difference.



(a) Planar (b) Convex (c) Concave

Fig. 2.6: Reflection mappings on different surface types. For planar and convex surfaces, a given scene point Q reflects into the viewpoint E in at most one surface point, while concave surfaces can provide multiple reflection points.

The idea behind our novel SSCR-approach builds upon these observations and is described in detail in Chapter 3.

2.3.3 Current mirroring methods in real-time rendering

Offline rendering techniques like raytracing and others are known to handle physically correct mirror reflections very well. In the following we will discuss current rendering techniques that address mirror reflections, focusing on methods that run at interactive frame-rates.

Mirror viewpoint

One of the simplest and most straight-forward methods for rendering planar mirrors is to mirror the viewpoint E and the view direction vector V around the plane containing the mirror surface and then render the whole scene from the perspective of this new mirrored viewpoint E' [DB97]. The resulting

viewport image can either be mapped onto the mirror geometry as a texture when rendering the original viewpoint's perspective, or be directly rendered to the viewport of E by transforming the point projected with respect to E' by an additional matrix multiplication. In the latter case, stencil buffers are often used to avoid mirror shading in parts of the viewport where no mirroring surface is present. In both cases, a geometrically correct projective mapping is applied to the scene visible in the mirror. On curved surfaces, this method is not applicable correctly anymore, since a simple perspective projection onto a curved surface introduces an error.

Environment Mapping

A simple technique that approximates reflections on curved surfaces and works fine in real-time is environment mapping [BN76][Gre86]. This method determines the color of a reflective pixel by a lookup in an environment map texture. The lookup coordinate is solely calculated by the reflection vector which is given by the viewpoint and the surface normal of the reflective pixel. The position of the surface point is neglected in this approach, meaning that the mapping error increases with the size of the mirroring surface. Generally, environment mapping can only produce a rough approximation of the mirroring process. The error vanishes if the mirroring object is very small and the objects in the scene that are approximated by the environment map are relatively far away from the mirror.

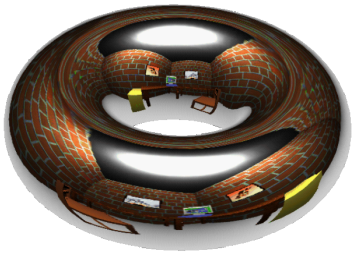


Fig. 2.7: Torus with view-independent environment mapping. Note that there are no interreflections on the inside of the torus, both the back inside and the front outside parts show the same mirror image. (Image courtesy of Heidrich and Seidel [HS98]).

On more complex surfaces, e.g. composed of several convex and concave surface patches and probable cuts in the continuity of the surface, environment mapping is not suitable for believable reflection rendering. Objects that contain both convex and concave regions can contain self-occluding parts that

lead to interreflections, not producible by environment maps. A similar problem is given by surfaces that do not have the topology of a sphere in the first place. Environment mapping also introduces a recognizable error on such objects, as seen in Figure 2.7.

Reflection subdivision

In 1998, Ofek and Rappoport introduced a method for rendering reflections on curved surfaces at interactive frame-rates [OR98]. Their approach maps a potentially reflected scene object to so called *virtual objects*. Those virtual objects represent a correct per-vertex description (i.e. a mesh) of the object's mirror image and can easily be rendered to the viewer's image plane by the conventional rendering pipeline. In order to compute the virtual object O' of a given scene object O, the scene object is tesselated to a mesh of sufficiently detailed polygon and vertex count. Then each vertex V of O is mapped to its *virtual vertex* V' reflected on the mirror surface, conserving connectivity information to reproduce the form of O'.

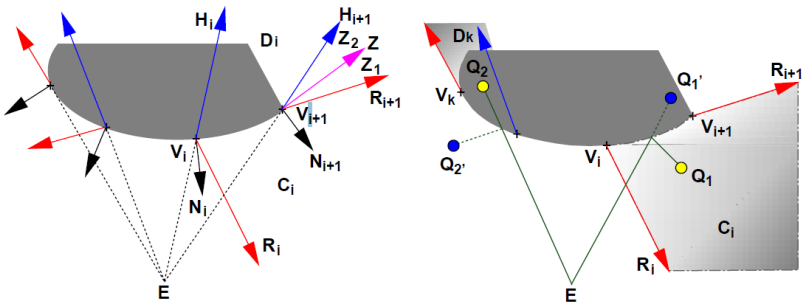


Fig. 2.8: Left: eye rays are reflected on the mirror surface resulting in the reflection vectors R_i (red) that enclose cells C_i. Right: a given scene vertex Q is transformed to a virtual vertex Q' by reflection in the cell's triangle on an tangential plane interpolated between the triangle's vertices. (Image courtesy of Ofek and Rappoport [OR98]).

In order to compute the position of the virtual vertex V' for a given scene vertex V, they introduced a space subdivision method called *reflection subdivision*. This method subdivides the space around a given mirror surface by planes spanned between the reflection vectors of the eye vector at each vertex of the mirror surface mesh. Each of those subdivisions of space is called *Cell*. This way, each triangle of the mirror surface mirrors exactly that part of the scene into the viewpoint that is bounded by the the triangle's cell. In order to avoid a facet effect on the mirror surface, the transformation of Q to

its virtual counterpart Q' is not simply performed by a mirroring of Q around the triangle's plane, rather the actual reflection point of Q in the triangle is represented in barycentric coordinates of that triangle. Those coordinates are then used as weights for an interpolation of the tangent plane at that reflection point with respect to the tangent planes of the triangle's vertices. Figure 2.8 illustrates this concept.

The challenging part of this method is finding the right cell (and thus the right mirror triangle) any given scene point Q lies in. For convex mirrors, each point of the space belongs to at most one cell. For concave or mixed-convex mirrors, they decompose the surface into pure convex and concave parts. To find the right cell for a given point Q on general convex or concave polygon reflectors, they proposed an associated approximate acceleration method, the *explosion map*. Basically, this map represents a circular image of the reflector's triangles that are visible to the viewer. The triangles are spherically mapped using a virtual sphere with an approximated center and radius that best describes the surface of the reflector. The mapping is performed in a way that every possible direction from the center of that sphere is represented by a map coordinate that lies inside a circle. Every triangle is rendered with a color corresponding to a unique ID of the reflector's triangles. Figure 2.9 illustrates such an explosion map for a simple convex polygonal reflector.

When searching the right triangle (respectively the cell) a scene point Q belongs to, Q is mapped to a lookup coordinate of the explosion map, making it possible to directly evaluate the ID of the triangle Q belongs to.

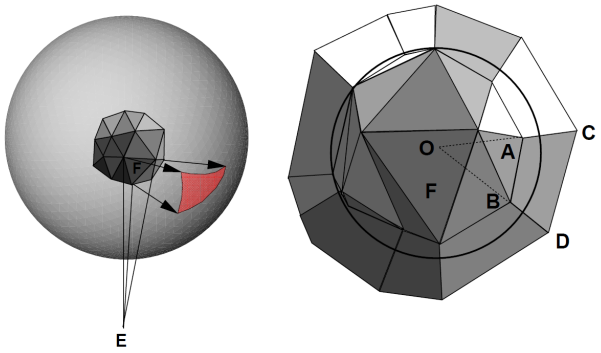


Fig. 2.9: Sample explosion map (right) for a polygonal reflector (left). Each reflector triangle is mapped to a virtual sphere approximating the reflector's surface. (Image courtesy of Ofek and Rappoport [OR98]).

Discussion This method achieves interactive frame rates as long as the number and complexity of the reflecting surfaces is not too high. The method

works fine especially with specific types of mirror surfaces like linear extrusions of planar curves. Arbitrarily convex reflector surfaces can be rendered well too using the explosion map. Due to their physical and reflective nature, concave reflectors are more difficult to handle in some situations, since they introduce mutually intersecting reflection subdivisions. For simplicity this method handles concave mirrors like convex ones, arguing that the error is acceptable since in such situations concave reflectors produce chaotic images anyway.

Since the computation of virtual objects is a mapping of an object's vertices to their virtual counterparts, only the vertices of an object are mapped according to the curved surface of the reflector. From there on, conventional triangle rasterization is executed, meaning that the triangle's edges do not correctly map to warped lines on curved reflectors. Therefore, a fine tesselation of the objects in the scene is necessary to achieve a certain degree of accuracy. Further, since general reflector surfaces are represented by simple polygon meshes, also those meshes would have to be tesselated in order to produce results of sufficient detail.

Sample-based cameras

Another, quite similar method for real-time rendering of curved reflections was given by Popescu et al. in 2006. They proposed *sample-based cameras* (SBCs) [PSM06], which are a collection of BSP trees that contain a number of pinhole cameras at their leaves to approximate the projection on a curved surface. This method too projects the scene geometry into their mirror image and then invokes the feed-forward pipeline to render the reflection images, thus facing the same potential need for tesselation in order to accurately visualize bent polygon edges on curved mirrors. The second point they have in common with Ofek's and Rappoport's method is that (on non-concave surfaces) the pinhole cameras too partition the space of their environmental scene.

This approach first generates a map of rays that are reflected by mirroring surfaces in the scene visible to the viewer. Depending on the number of desired mirror bounces, there are first-order, second-order and more such ray maps for a particular scene. Based upon this ray map, they then build up a hierarchy of neighboring rays sharing the same reflection history, i.e. the same reflection path among a number of reflectors. Such a group of rays is called *reflection group* and – over several reflection steps – transports continuously projected scene information to the viewer. This projection is defined by a reflection group projection function, which is approximated by several pinhole cameras whose frustum encloses the rays of the reflection

group. When rendering reflections, a given scene point is handled by the camera that contains the point in its frustum.

In order to create the SBC for a scene, for each of these reflection groups a BSP tree of cameras is set up. This BSP tree hierarchically subdivides the reflected frustum of a reflection group on a curved surface, and contains cameras at their leaves. The subdivision decomposes the frustum into further hierarchy levels of the BSP tree as long as the error given by a single camera is greater than some desired measure of accuracy. Figure 2.10 shows a BSP tree for an example reflector. In order to project a scene point onto a reflector, the BSP tree is traversed to find the corresponding camera containing p. For multiple reflection bounces the first reflection point is projected forward the same way iteratively.

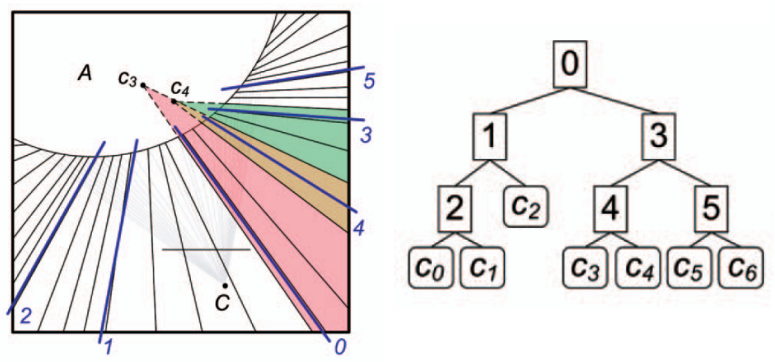


Fig. 2.10: Example for a SBC. For a convex reflector observed by C, reflection rays are computed. They then are hierarchically grouped into a BSP tree of certain depth. At its leaf nodes, simple cameras are attached whose frustums are bounded by the reflection rays along the binary path to the leaf in the BSP tree. (Image courtesy of Popescu et al. [PSM06])

Discussion Although this method achieves interactive curved reflections comparable in accuracy to raytracing, it has some drawbacks. Since their approach is based on a binary partitioning of space into a number of pinhole camera frustums, concave reflectors are not supported. For concave reflecting surfaces, parts of the environment space are contained by many reflection groups, and their rays do not form a frustum intersecting at a pinhole camera position anymore. Further, similar to Ofek and Rappoport, they primarily address mostly continuous reflecting surfaces with not too much complex geometrical detail.

2.4 Real-Time Global Illumination in Point Clouds

2.4.1 Background to our GI Renderer

The base for this thesis is given by an earlier work in the *Terapoints*-Renderer, namely a real-time global illumination (GI) renderer for point clouds [Pre10]. This GI algorithm distributes *Virtual Point Lights* (VPLs) in the scene, depending on the type and situation of the light source. VPLs were introduced by Alexander Keller in his Instant Radiosity technique in 1997 [Kel97]. These VPLs serve as an approximation of the radiosity of an illuminated surface region, and are used to calculate the indirect illumination of the scene.

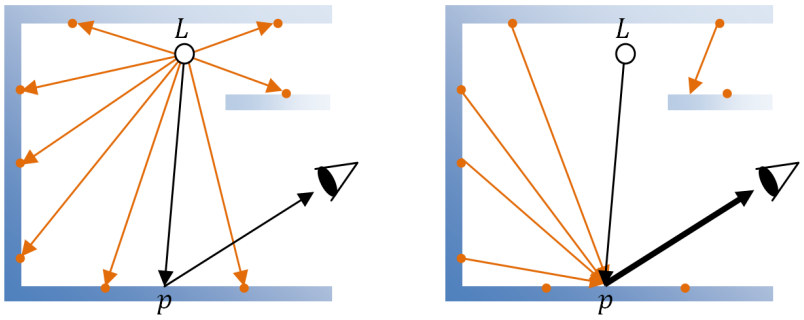


Fig. 2.11: Left: Based on the light source, a number of VPLs is seeded over the scene. Right: Considering correct visibility for those VPLs, a scene point p is indirectly illuminated by some of the VPLs' reflected light, resulting in more accurate information transported to the eye than using only direct illumination.

In order to use correct visibility information for those VPLs, the GI renderer uses *Imperfect Shadow Maps* (ISMs), as proposed by Ritschel et al in 2008 [RGK+08]. ISMs represent a powerful method for fast visibility-approximation for a high number of VPLs. In principle, such an ISM is a low resolution shadow map, encoding depth information of the whole hemisphere around a VPL in a paraboloid map [BApS02]. As we have to store one ISM for each single VPL, all these low-resolution maps are combined in one huge ISM buffer. The radial depth image represented by an ISM is obtained by fast splatting of the scene points onto the map, choosing a splat size perspectively decreasing with increasing distance between the VPL position and the position of the splatted point. In fact, while conventional mesh scenes have to perform separate sampling step in order to obtain point samples from their polygon objects for ISM splatting, the GI algorithm implemented in our point cloud renderer exploits the fact that the scene data can be used for ISM creation "as is", since it is already represented by points.

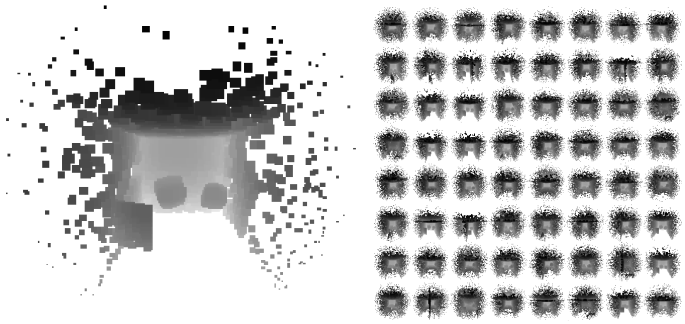


Fig. 2.12: Left: Sample ISM Buffer with an resolution of 256x256 pixels. Right: combined ISM Buffer containing the ISMs of 64 individual VPLs.

Figure 2.12 shows an example ISM and combined ISM buffer. As Figure 2.13 points out, VPL visibility is vitally important for GI rendering, as it adds indirect shadows to a scene, which contributes a great deal of realism in the result.

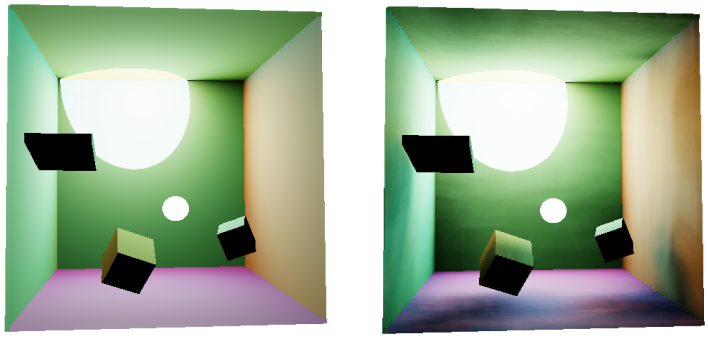


Fig. 2.13: Comparison of the appearance of a demo sene without VPL visibility consideration (left) and with VPL visibility test using ISMs (right) clearly enhancing the realism by introducing indirect shadows.

Shading is performed per-pixel in screen space, i.e. only for those parts of the scene that are visible to the viewpoint. However, applying indirect illumination calculations to every pixel requires various world-space information to be available for each single pixel. In fact, the shading system uses a so called *Camera G-Buffer* that is rendered once per frame for the current viewpoint, which has the size of the viewport and containing all data of the scene surface point associated with that pixel necessary for indirect illumination calculations. This G-Buffer consists of the following data, which is partially stored in a compressed way, and distributed over several textures:

- Material information
  - Diffuse reflection color
  - Specular reflection factor
  - Specular power (shininess)
- Geometry information
  - world-space surface normal
  - world-space position of the surface point (view-space depth)

Figure 2.14 shows an example for a Camera G-Buffer that is rendered for a particular GI scene. In Chapter 3 we will reuse the surface normal information stored in the Camera G-Buffer that was already rendered for a frame for conventional GI calculations, and setup our novel SSCR-algorithm on this data.

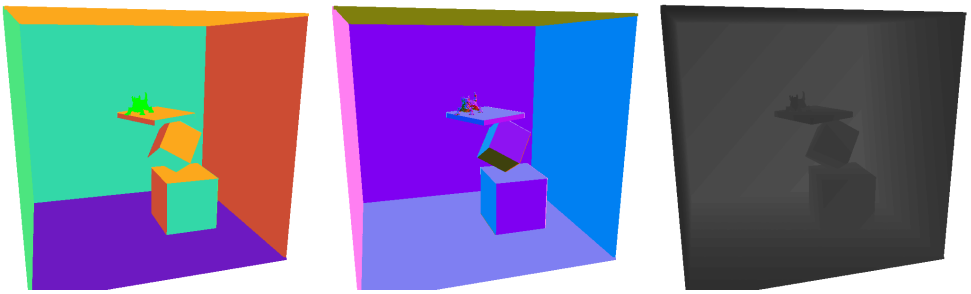


Fig. 2.14: RGB visualization of the three RGBA-textures of a Camera G-Buffer for a demo scene in our point cloud GI-Renderer. The left texture contains surface color, the middle texture surface normals and the right stores linear depth to reconstruct world-space position. Note that the alpha-channel not visible in this figure stores further information (shininess and specular intensity of the surface)

When shading each single pixel in the fragment shader, its G-Buffer data can be looked up, and indirect illumination calculations are performed with respect to the VPLs which illuminate the surface associated with that pixel. Each pixel is illuminated only by a subset of the available VPLs. In order to optimize performance even further, each VPL only illuminates a subset of the frame buffer pixels by applying interleaved sampling (Keller et al [KH01]). While executing this procedure, indirect illumination values are accumulated, resulting in an indirect illumination image of the scene that converges towards an optimal quality as the number of VPLs are increased. The final image is

achieved by adding direct illumination to this accumulated image and tone-mapping the result.

With this method, we are able to achieve GI shading in our point clouds scenes simulating several indirect light bounces at real-time framerates. There are some restrictions in the ability of correct illumination simulation though. As for every many-point illumination technique, our GI implementation achieves best results in scenes with a high component of diffuse reflecting surfaces. Specular reflecting objects can be rendered with low error as well, as long as the specular intensity of a surface material does not become to high. If so, the discrete VPL sampling can lead to aliasing artifacts in form of light sparkles appearing in the image (see Figure 2.15). This dues to the problem of undersampling a radiating area by a finite number of VPLs in order to approximate its radiance.

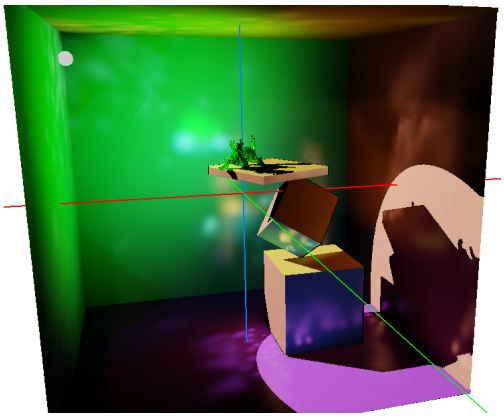


Fig. 2.15: Scene rendered with global diffuse intensity 0.5, specular intensity 0.95 and shininess 1000. At the highly glossy surfaces, the distribution of VPLs lead to the appearance of sparkles.

A simple way to avoid these light sparkle artifacts is to clamp the maximum light contribution of a VPL, conceding the introduction of a bias. Hašan et al [HKWB09] address the problem in a more sophisticated way, replacing VPLs by *Virtual Spherical Lights* (VSLs). Although their method is not designed for real-time application, they are able to successfully eliminate such illumination spikes.

2.4.2 Enhancing vizualisation power

In this thesis, we improve the current GI render mode implemented in *Terapoints*, by adding perfect reflections (mirror reflections) to our scenes. Mirror reflections are an omnipresent effect in many real-life scenes. Even though

people may be confronted with many observations of mirror reflections in daily life they are not aware of, they are still able to sensibly recognize an error in an artificially created image where those effects are wrong or missing. This thesis introduces a novel approach for rendering mirror reflections at interactive rates, even on highly detailed curved surfaces.

Since our current GI implementation approximates each possible, arbitrarily complex surface reflection behavior by reducing them to a weighted combination of Lambertian-diffuse and Phong-specular reflections, the method is not able to reproduce perfect mirror reflections properly. We therefore have to complement our current algorithm by a method that handles such perfect reflections. The following chapter shows, how our novel technique performs this task.

Chapter 3

Screen-Space Curved Reflections

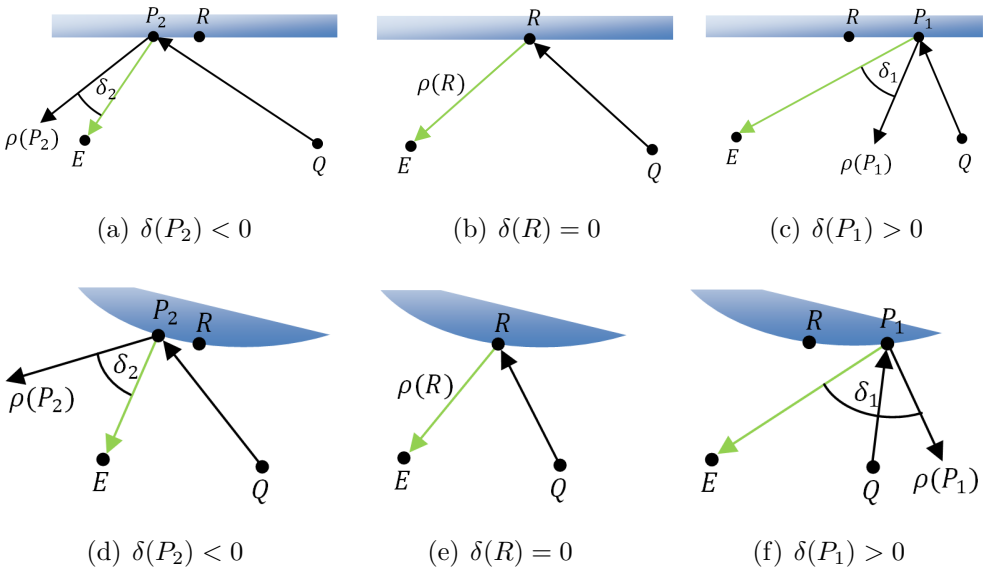
In this chapter we present our new method, *Screen-Space Curved Reflections* (SSCR), which simulates mirroring reflections on almost arbitrarily complex surfaces at interactive frame rates. The technique is designed for the implementation in huge point cloud scenes, but not restricted to this geometry representation, as Chapter 6 describes later on.

3.1 Basic Idea

Our approach pursues the following idea: Given the complete point-cloud data and an image of an observer's projected view of a scene containing mirroring surfaces, we want to find for each scene point the pixel in the viewport that reflects the point into the eye. Of course, testing each point in the scene against each pixel in of the viewport is out of the question. In order to achieve our goal, we build on the following important observation:

Consider a scene containing an eye-point E, a scene point Q and a continuous mirror surface M. Each point P on the mirror surface M provides two vectors: an eye-vector $\overrightarrow{PE}$ and a reflection vector that indicates the reflection direction of an incoming light vector $\overrightarrow{QP}$. Looking at the change of orientation of this reflection vector relative to the eye-vector, we observe that the deviation of this reflection vector from the eye-vector changes continuously while moving the reflection point P over the surface. More precisely, this deviation shows a different sign based on the position of P relative to a ideal reflection point R, where the deviation is zero. Figure 3.1 illustrates this observation for both pure planar and pure convex surfaces. For concave surfaces this holds too, although they represent a slightly different case.

Based on these considerations, in the following sections we will formulate a continuous function $\delta(P)$ over a surface, that has the useful characteristic that it contains the positions of the surface's reflection points at its roots. With the availability of such a function, we are able to perform a root finding in order to determine the surface reflection point for a given scene point. In



(a) $\delta(P_2) < 0$ (b) $\delta(R) = 0$ (c) $\delta(P_1) > 0$

(d) $\delta(P_2) < 0$ (e) $\delta(R) = 0$ (f) $\delta(P_1) > 0$

Fig. 3.1: Illustration of the dependence of the deviation between reflection vector and eye-vector on the location of some surface point P on planar (upper row) and convex (lower row) reflectors. The deviation is indicated as angle δ between both vectors. We observe that the absolute value changes continuously while sweeping P over the surface, being zero at the reflection point R. The sign of $\delta(P)$ indicates the relative location of P to R.

SSCR, mirror reflections in point clouds are rendered by executing this root finding algorithm in screen-space for each point of the scene and splatting these points onto the framebuffer.

Sections 3.2 and 3.3 introduce the before mentioned function, which is named *mirror-space error function*. We will first formulate a new affine space called *mirror-space* for better illustration of the function characteristics. Then we will evaluate the characteristics of this function. First we will observe its behavior for the three simple major types of surface curvature (planar, convex and concave surfaces), and then for arbitrary, more complexly curved surfaces.

In Sections 3.4–3.4.1 we will discuss possible ways to find a reflection point on the three before mentioned simple curvature types. Based on the fact that a reflection point is always settled at the root of the error function, we will introduce a fast approximate 2D root finding algorithm that operates in screen space in order to efficiently produce our point splatted mirror images.

In Section 3.5 we will then extend our view to general, higher-complex mirror surfaces. In this section we will introduce a method that segments the mirror surfaces in the framebuffer into a number of regions of homogeneous

curvature, each being either planar, convex or concave again. After this segmentation, we are able to distribute the available scene points between the resulting regions and apply our screen-space root-finding algorithm to them.

Finally, in Sections 3.6 and 3.7, we will further develop our algorithm. We will give a solution on the problem of visibility of the mirror images in SSCR, and discuss several techniques in order to enhance its efficiency and image quality.

3.2 The mirror space

For a general formulation of the error function in the following section, we define a new coordinate space called *mirror space*. Similar to the tangent space, the mirror space is defined over a point of a surface. While the tangent space is parameterized by the surface point (u, v) and its normal N_{uv}, the mirror space is parameterized by (u, v) and the camera parameters (viewpoint E and view-coordinate system) of the viewer. The mirror space coordinate system in a given surface point P is set up in a way that the world-space eye-vector $\overrightarrow{PE}$ coincides with the mirror-space z-axis, and the x-axis of the mirror-space coordinate system lies coplanar to the x-axis of the view-coordinate system (see 3.2). Note that the latter is not essential for the nature of this space, but is chosen for convenience. We will refer to these characteristics later.

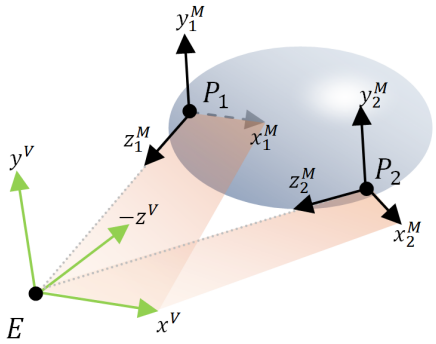


Fig. 3.2: Mirror-space coordinate systems of two surface points P_1 and P_2 on an oval reflector. The green coordinate system is the view-space system. Each surface point defines its own mirror-space coordinate system with z-axis pointing towards the viewpoint E. Note that the mirror-space coordinate system is always rotated in a way that its x-axis lies coplanar to that of the view-coordinate system, illustrated by the planes they span.

3.3 The mirror-space error function

3.3.1 Formulation

Based on the previous observations on the relation between surface point and reflection deviation illustrated in Figure 3.1, we will now formulate a continuous function over a homogeneously curved surface that expresses this deviation for a given scene point Q and eye-point E by a signed error angle. For 1D surfaces, the function evaluates to one error angle, while on 2D surfaces it provides two angles. We will first observe the 1D case.

To make use of the observations in Figure 3.1, the error angle has to provide a sign in order to indicate its relative position to the reflection point. Formulating the error function as the arcus cosine of the dot-product between the reflection vector and the eye-vector would would be insufficient, since the dot-product between these two vectors doesn't preserve their relative position to each other, thus not providing a sign for the angle.

We therefore first formulate a vector valued function that expresses the deviation by a difference vector between the reflection vector ρ and the eye-vector $\overrightarrow{PE}$:

$$\vec{\delta}_{QE}(P) = \rho(\overrightarrow{QP}, N) - \overrightarrow{PE} \tag{3.1}$$

Equation 3.1 evaluates a world-space vector, by whose length we can determine the degree of deviation. So far, this vector does not give us any information about the relative position of the reflection point R in P. We therefore transform this vector $\vec{\delta}$ into mirror space. Since the mirror-space z-axis coincides with the eye-vector $\overrightarrow{PE}$, it separates the vector space in P in two halves of different error sign. Figure 3.3 illustrates this for the 1D planar case. Transforming $\vec{\delta}$ into mirror space thus yields a vector whose x component's sign indicates the relative position of a reflection point R.

After mirror-space transformation of $\vec{\delta}$ and substitution of the reflection function for ρ, we obtain the following formulation:

$$\vec{\delta}_{QE}(P)^M = \overrightarrow{QP}^M - 2N^M(N^M \cdot \overrightarrow{PE}^M) - \overrightarrow{PE}^M \tag{3.2}$$

The superscript M indicates the mirror-space transformation of the vectors. For 1D surfaces, the resulting vector $\vec{\delta}^M_{xz}$ is a 2D vector whose x component encodes the relative position information in its sign. We finally calculate the arcus tangent of $\vec{\delta}^M_x / \vec{\delta}^M_z$ and obtain a signed error angle for the error along the x axis. For 2D surfaces, we have a 3D vector $\vec{\delta}^M_{xyz}$, by which we calculate two error angles, one for the x and one for the y mirror space direction. This results in the final formulation of the *mirror-space error-function*:

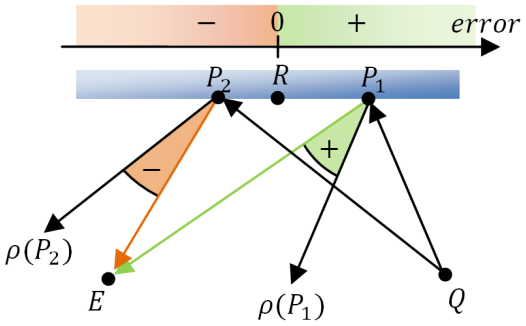


Fig. 3.3: Separation of the surface in two regions of different error sign. At the reflection point R, the error is zero. From each point P, the direction in which to find R can be determined by the sign of the error angle.

$$\delta_{\{x|y\}}(P) = \Phi_{\{x|y\}}\left(\overrightarrow{QP}^M - 2N^M(N^M \cdot \overrightarrow{PE}^M) - \overrightarrow{PE}^M\right)$$
$$with \quad \Phi_{\{x|y\}}(\vec{v}) = \arctan\frac{\vec{v}_{\{x|y\}}}{\vec{v}_z} \tag{3.3}$$

With Equation 3.3 we now have a continuous error function over a surface that expresses the deviation of the reflection vector $\rho(P)$ to the eye-vector $\overrightarrow{PE}$ for a given surface point P by a signed error angle (in 2D, two signed error angles). In the following, we will analyze the characteristics of this function for 1D surfaces. Later we will extend our evaluation to 2D surfaces, which is the actual interesting case for our 3D scenes.

3.3.2 1D surfaces of basic curvature

We will now analyze the characteristics of the δ function for the three basic types of curved surfaces (planar, convex and concave) in 1D. We can state the following properties of the function:

1. On continuous surfaces (containing no edges or trenches), the mirror-space error function is continuous. This is due to the fact that both its terms $\rho(\overrightarrow{QP}, \mathrm{N})$ and $\overrightarrow{PE}$ are continuous vector valued functions on such surfaces, and thus their difference is.

2. If the surface contains a reflection point R for a given scene point Q and viewpoint E, then the function evaluates to zero at that point, i.e. $\delta(R) = 0$.

3. The function is axially symmetric in Q and E, meaning that interchanging Q and E yields the same value for δ:

$$\delta_{QE}(P) = \delta_{EQ}(P) \tag{3.4}$$

For $P = R$, this is trivial. Figure 3.4 illustrates this fact for a general P, Q and E.

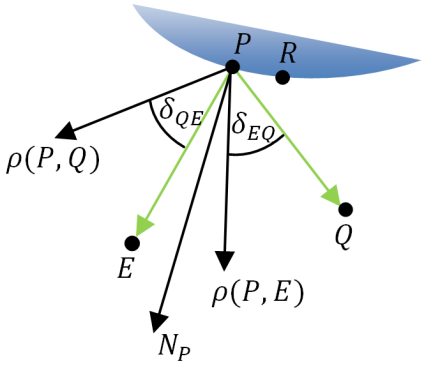
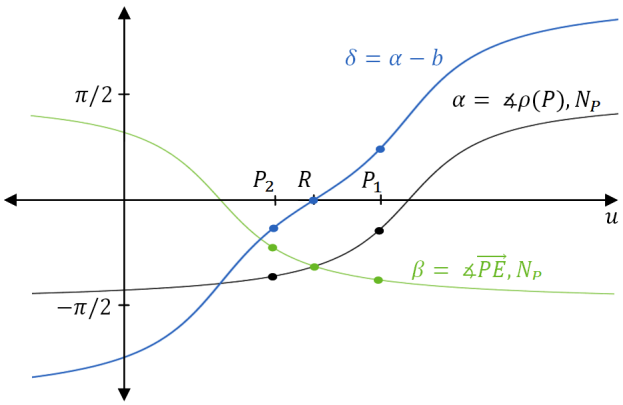


Fig. 3.4: Symmetry of the mirror-space error function.

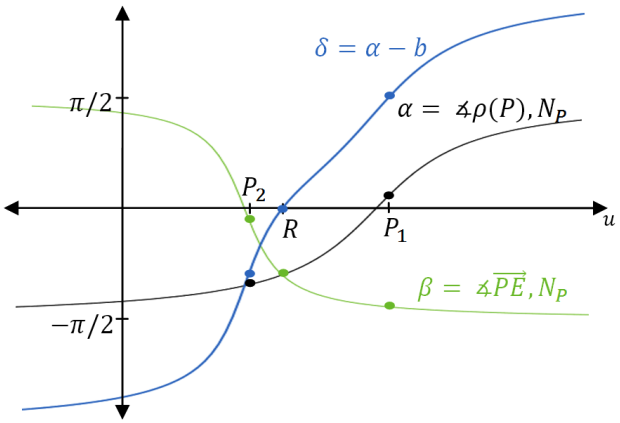
4. Considering only the interval of the surface that is visible to the eye-point, for homogeneously convex and planar reflectors the function is strictly monotonic. This is shown by figures 3.1 and 3.5 At concave surfaces this is not guaranteed anymore, as explained later.

5. From (2) and (4) it follows that for convex and planar reflectors, the sign of the function value of $\delta(P)$ always changes around R (see Figure 3.3).

Planar and convex surfaces Figure 3.5 plots the δ-function for the planar and convex surface examples shown in 3.1. It also plots two further functions α and β, which both are angular difference functions over the reflector surface parameterized by u. Lets call these two functions *inner functions*. α is a function of the difference angle between the surface normal N_P and the reflection vector ρ, while β evaluates the difference angle between N_P and the eye-vector. In these plots, δ is defined by the difference between these two functions. Note that this formulation of the error function is equivalent to the previous one. For each function, the local change of the difference angle is determined by its first derivate. In the convex surface plot, the slope of the β-function is stronger compared to the planar case, due to the



(a) planar



(b) convex

Fig. 3.5: Plot of the angular δ-function and its inner functions for the planar and convex surfaces in Figure 3.1.

fact that the eye-vector deviates more quickly from the surface normal as the surface bends back (see also Figure 3.1). The α function however stays approximately the same in comparison to the planar surface. This is due to the fact that the reflection function ρ changes with the surface normal N_P, i.e. it too is a function over the surface.

Considering only the surface interval *visible to* E, in both directions $+u$ and $-u$, α and β converge to $-\frac{\pi}{2}/+\frac{\pi}{2}$, and thus δ to $-\pi/+\pi$. (Evaluating the function for the convex case over a complete closed surface like a sphere, there is another hypothetical reflection point R on the for E hidden backside of the sphere, reflecting an incident vector with angle $|\theta_i| > \pi$).

Since we stated that within this interval in both the planar and the convex case the function is strictly monotonic, we therefore see that there exists at most one root for δ in the visible surface interval. This root is the reflection point R. Further, for both cases the sign of the function value of δ always changes around R. Thus, R separates the surface M into two regions, one with positive, the other with negative error.

Concave surfaces Concave reflectors show different characteristics in the δ-function. Their main difference to planar or convex reflectors is that they can provide multiple reflection points R_i.

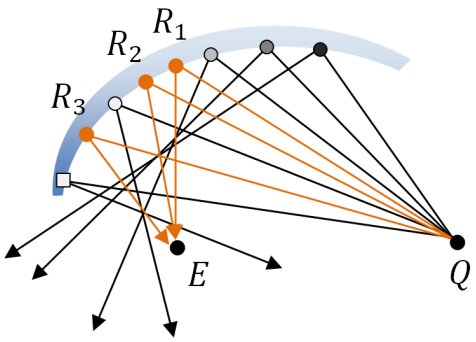


Fig. 3.6: Concave reflector providing three reflection points. The gray-code of each surface point indicates the mirror-space error (i.e. the absolute value of δ), the shape indicates its sign.

Figure 3.6 illustrates such a concave surface. Looking at the δ-function for this reflection, we see that there are three roots (see Figure 3.7). The angular difference β between eye-vector and surface normal here oscillates around approximately $-\frac{\pi}{8}$. This is due to the viewpoint E being relatively close to the surface and the focal point in this example, i.e. the concave reflector surface and its normal kind of orbit E, resulting in an approximately even balance between surface normal N_P and eye vector E along the surface.The α-function describing the angular difference between the reflection function ρ and the surface normal N_P also shows a more complex behavior for this setup. Generally, both curves strongly depend on the location of E (respectively Q, referring to Equation 3.4) relative to the concave surface.

The setup shown in Figure 3.6 represents the actual difficulty of concave surfaces, namely the case when the concave mirror produces chaotic, extremely warped images of the environment, where conventional mirror mapping of a tesselated scene object for subsequent feed-forward rendering is not reasonably applicable anymore. [OR98] mention this case. Their

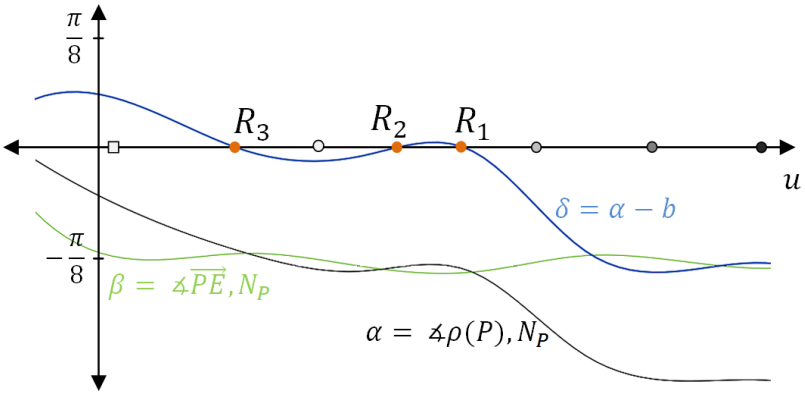


Fig. 3.7: Function plot of the angular δ-function and its inner functions for the concave reflector shown in Figure 3.6. (Note: This plot is an approximation given by a cubic spline through the sample points).

method subdivides the space in front of a concave mirror in three regions: A) in-between the surface and the focal point, where the mirror enlarges the image, B) in front of the mirror *and* its focal point, where the viewer observes an upside-down image of the scene, and C) the region around the focal point, where [OR98] state that reflection is *"unpredictable and chaotic anyway"*. Therefore, they treat these cases as if dealing with convex surfaces, settling with any arbitrary result. [PSM06] on the other hand do not support concave reflectors, thus avoiding this problem altogether.

A maybe more common situation would be the eye-point being further away, resulting in a flipped mirrored image. In this case, the δ-function provides only one reflection point, and its curve is more comparable to the convex case. Therefore, the concave mirror provides a recognizable mirror-mapping of its environment. Figure 3.8 shows an example point cloud scene containing a mirroring sphere (convex) and a parabolic mirror (concave) rendered with GI and our new SSCR algorithm. Note that in comparison to the sphere, the concave mirror flips the image upside down and left to right.

3.3.3 2D surfaces

In 1D surfaces, the mirror-space error function can exactly determine the position of a reflection point R (if existing) relative to some surface point P. On 2D surfaces we deal with a 3D mirror-space coordinate frame. The difficulty for 2D surfaces is the proper choice of the orientation of the mirror-space coordinate axes. In a 1D surface, we deal with one root point separating the surface into points of different error signs. 2D surfaces in contrast are

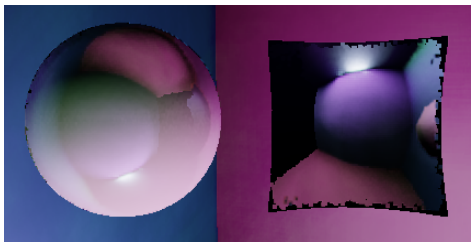


Fig. 3.8: Example point cloud scene rendered with GI and our new SSCR algorithm. The mirroring sphere on the left contains a diffuse reflection component, while the concave surface on the right is a pure mirror. Note that the artifacts at the borders of the parabolic mirror are the results of discontinuities of the surface due to front facing point splats.

separated into regions of different error signs by two root *lines*, one line for each dimension of the surface parameter space. Each root line separates the surface in two regions of different error sign. Figure 3.9 shows an example for the root lines in a 2D surface.

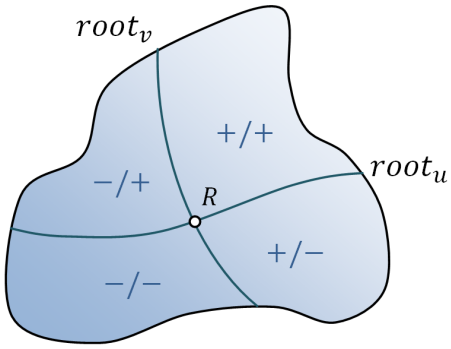


Fig. 3.9: Schematic illustration of the root lines of a 2D mirror-space error-function. Each line separates the surface with respect to one parameter dimension into two regions of different error signs, yielding 4 error sections. At the intersection point of the lines, the error in both surface dimensions is zero, thus there we find a reflection point.

The actual reflection point R is now located at the intersection point of these lines. The problem is that the location and orientation is just as unknown as the location of the root point in the 1D case. In the 2D case, the mirror-space error function evaluates to two difference angles δ_x and δ_y. If we assume the two root lines of a surface to be approximately aligned with the mirror-space coordinate axes, we can determine the 2D-direction in which to look for R by an individual evaluation of the error sign of both the x and the

y component of δ. In the following development of our algorithm, we will build on this assumption, admitting that this simplification may not work best for all surface types. However, we will show that this simplification is acceptable in order to develop a fast and efficient mirror rendering algorithm.

3.3.4 Complex surfaces

So far, we have discussed the mirror-space error function only for the three basic types of surface curvature (planar, convex, concave). We stated that for planar and convex surfaces there is at most one reflection point, while in some cases concave surfaces can even provide multiple reflection points.

In general, surfaces can show an arbitrarily complex curvature, containing several surface parts of different curvature. On such surfaces, we can find multiple reflection points. The error functions of such surfaces can contain several extrema, and therefore several roots. The mirror-space error function is not necessarily strictly monotonic anymore. Common surfaces can also contain edges or trenches, which even breaks the continuity of the mirror-space error function. Figure 3.10 shows an example of an arbitrarily complex 1D surface and its mirror-space error function.

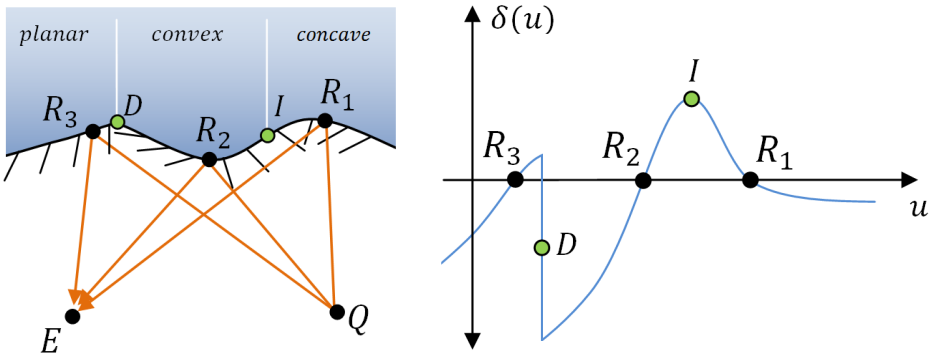


Fig. 3.10: Example of an arbitrarily complex 1D surface. The reflection vectors of Q along the surface are indicated by the small direction lines. D represents a point of discontinuity, I an inflection point of the surface curvature.

In this figure, the surface can be partitioned into different intervals of one of the aforementioned three basic types of curvature. Planar and convex intervals can contain at most one reflection point, while concave ones can also provide multiple. The borders of these partitions are given either by *discontinuity points* (D) or *inflection points* (I) of the surface. Discontinuous

surface points result in discontinuous steps in the mirror-space error function, while inflection points represent function extrema.

On 2D surfaces of higher complexity, we can also find several reflection points. In such cases, the surface contains multiple root lines, and therefore multiple intersection points of different lines. Figure 3.11 gives an example for such a surface. In the following, we will discuss, how we can make use of the function δ in order to find the reflection point on a mirror surface for a given scene point.

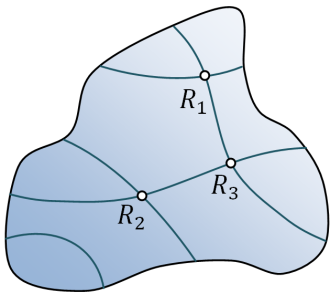


Fig. 3.11: Example of an complex 2D surface containing multiple reflection points, which are the intersection points of several root lines.

3.4 Finding the reflection point

In this section, we will develop an algorithm for efficiently finding a reflection point R on a mirror surface. The algorithm is restricted to continuous surfaces of homogeneous curvature, i.e. pure planar, convex or concave surfaces. Later we will show how we manage to render surface reflections on arbitrarily complex reflectors by partitioning their surface into elementary parts of such homogeneous curvature.

3.4.1 2D root finding in mirror space

Equation 3.3 defines a mirror-space error function, which gives two signed error angles δ_x and δ_y on the surface in mirror space. Based upon the signs of these deltas, we can approximately infer the relative direction in which a reflection point R is located in each mirror-space dimension. As already mentioned, this assumes the mirror-space coordinate frame to be roughly aligned to the root lines (i.e., the lines where δ_x and δ_y are zero) of a surface.

The actual direction in which to find R on a surface in 2D can therefore be estimated by following the individual 1D directions indicated by δ_x and

δ_y. Figure 3.12 gives an example of this. In this figure, the point P lies in a region of positive δ_x and negative δ_y. Based on the sign of these two deltas, we can approximately estimate the direction of R.

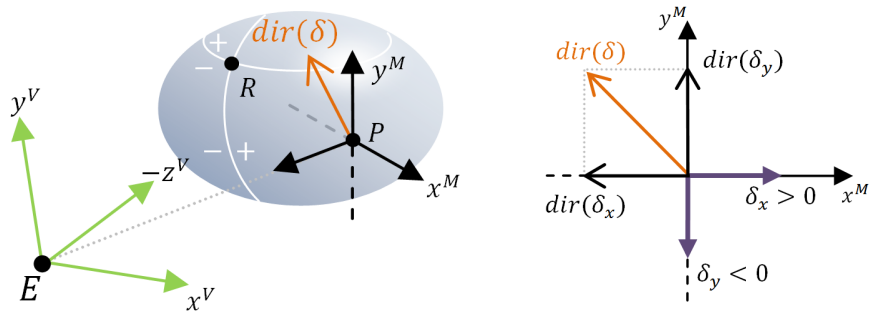


Fig. 3.12: Estimation of the 2D root direction by decomposition into an individual x and y direction, both of which can be determined by the two mirror-space errors δ_x and δ_y. The white lines on the reflector indicate some (arbitrarily chosen) root lines.

Note that, in order to follow the direction of R on the surface, the mirror-space coordinate axis is not always aligned in a way that its x and y axis perfectly match the tangential plane of P. Since the orientation of the z-axis is independent from the surface normal, but solely determined by the direction to E, the mirror-space coordinate system of a point P can be rotated away from this tangential plane, as illustrated in Figure 3.13. However, this rotation can never exceed 90° for both the x and y direction, because in this case the camera would observe the surface point from its backface. Thus, this rotation does not affect the direction that has to be chosen based on the sign of δ. It is therefore valid to choose the direction to go on the surface (tangent-space), solely upon the delta-vector that is given in mirror-space.

Since the mirror-space error function contains the reflective surface point at its root, we can perform a root finding in order to converge P to R. What we want is a fast procedure to find the reflection point within a local surface area of homogeneous curvature. We therefore need a fast 2D root-finding algorithm, i.e. an algorithm that finds the point on the surface where both δ_x and $\delta_y = 0$. The algorithm should provide a good success rate (with respect to possible misses that result from certain simplifications we make for the benefit of performance).

In the following we will develop an efficient 2D root finding algorithm that operates in screen space, i.e. directly on the pixels of our framebuffer. Our algorithm approaches the 2D root finding problem by decomposing it into two

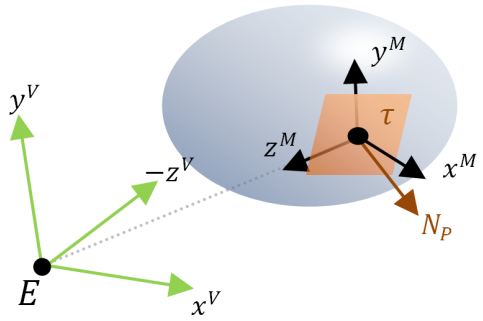


Fig. 3.13: Relative rotation between mirror-space and tangent-space coordinate system. τ shows the tangent plane, and N_P the surface normal in P. For visible points, the rotation of the mirror-space coordinate system can never exceed 90°.

1D root finding tasks, which can be evaluated simultaneously. Originating from some random start position P on the surface, our algorithm tries to approach the reflection point R by stepping in a direction that estimates the direction of R by an individual evaluation of the local mirror-space errors δ_x and δ_y.

First we will evaluate the relation between a reflection point search in mirror space and in screen space. Afterwards we describe our 2D screen-space root-finding algorithm and discuss its strengths and weaknesses.

3.4.2 Mapping mirror space to screen space

We have defined the mirror space as always being oriented towards the viewpoint and providing a coordinate frame that is x-axis aligned to that of the view space. Due to this definition, there is a strong correspondence between mirror space and view space, and thus screen-space. In this section, we will discuss this relation.

Let M be a continuous mirror surface of *homogeneous curvature*, i.e. pure planar, convex or concave, in the view-space system of an observer E. When perspective-projecting M onto the image-plane of E, the resulting projected image conserves neighborhood information of each surface point P of M. Note that for self-occluding concave surfaces, this also holds, because since we stated that the surface has a homogeneous curvature, its self-occluding part would only reduce the projected part of the surface visible on the image-plane, but not produce neighboring viewport-pixels associated with non-neighboring surface points.

Further, the projection of the mirror-space coordinate system in an arbitrary point P has the property that its x-axis lies parallel to the normalized

device coordinate system of the camera. The latter is due to the definition of the mirror space given in Section 3.2, aligning its x-axis in world space so that it lies coplanar to the view-space x-axis (see Figure 3.2).

Since the mirror-space z-axis is always oriented towards the viewpoint E, its projection defines a single pixel (i.e. it is neutral for x and y in device coordinates). Examining the projection of the mirror-space y-axis to the view plane, we see that it generally results in a slightly inclined vertical line in the viewport. This is due to the tilt of the mirror-space y-axis as the z-axis vector rotates towards the eye-point when shifting away from the axial centers of the image plane. This incline though is relatively small for commonly used view frustums. It is biggest for surface points observed along the corners of the view plane, and zero on its central horizontal and vertical axes. Figure 3.14 illustrates the angular tilt of the projected mirror-space y-axis along the diagonal of the image plane for 90°, 120° and 150° vertical fields-of-views (FOVs). We see that the tilt converges towards a 45°-rotation. With increasing FOV, the error extends more and more into the screen center. The 90° FOV most commonly used in computer graphics has an acceptable maximum tilt of approximately 25°. Note that this tilt error is independent of the view plane's aspect ratio.

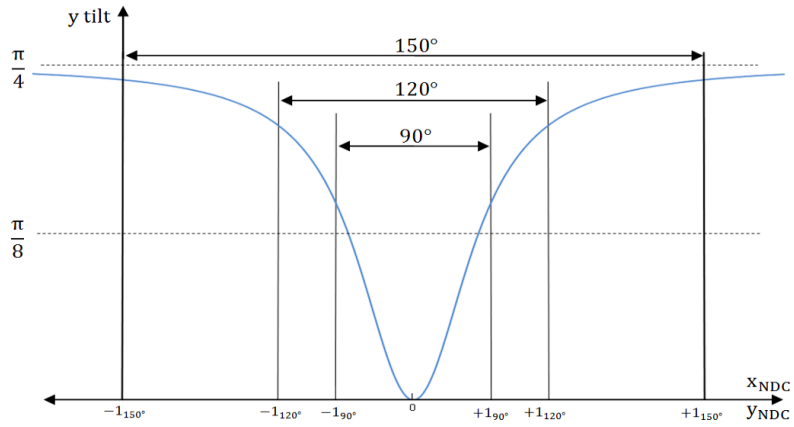


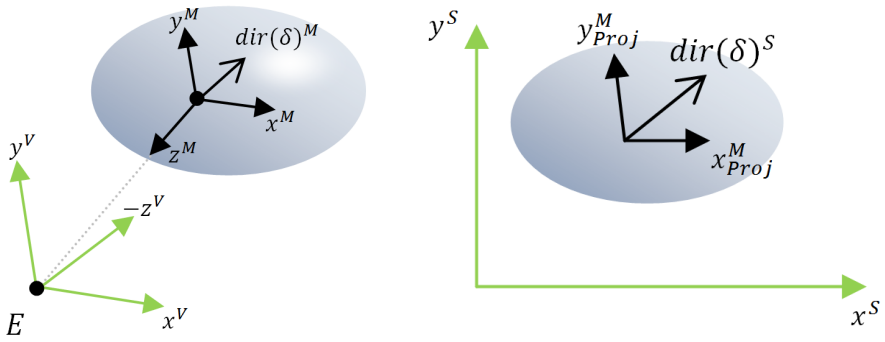
Fig. 3.14: Dependence of the projected mirror-space y-axis tilt from the projected position of the surface point P and the vertical field of view of the camera. The vertical borders indicate the error regions incorporated by the different FOVs.

Of course, alternatively we also could have defined mirror-space by aligning its y-axis coplanar to that of the view-space system, accepting a tilt in the x-axis projection. But since most commonly used camera models use a wider horizontal FOV than a vertical one, we would deal with a bigger error that way.

We recapitulate the following facts:

- The rotation of the mirror-space coordinate system in relation to the tangent plane doesn't affect the determination of the direction to the reflection point on the surface.
- The projection of a homogeneously curved surface conserves point neighborhood in pixel-space.
- The projection of the 3d mirror-space coordinate system of a surface point P is nearly a perfectly axis-aligned 2d coordinate system in screen-space.

Considering these, we can state a strong correlation between (u, v)-space, mirror-space and screen-space x- and y-directions, as illustrated in Figure 3.15. This means that while searching a root, stepping into – from the camera's point of view – positive x-direction in *mirror-space* directly corresponds to going into positive x-direction in *screen-space*. Similarly, going into positive y-direction in mirror-space corresponds to a step into positive y-direction in screen-space. Note that as already mentioned, this correlation introduces a minor error due to the slightly warped projection of the mirror-space coordinate system onto the view-plane. However, we have observed that this error is not objectionable.



(a) mirror-space direction $dir(\delta)^M$ (b) screen-space direction $dir(\delta)^S$

Fig. 3.15: Illustration of the correspondence between mirror-space direction and screen-space direction.

Due to this observation, we are now able to perform the root finding on homogeneously curved surfaces in *screen-space*, i.e. directly on the pixels of the viewport image that shows the scene from the perspective of the viewer. This has several advantages:

- In screen space, we only find reflection points that are actually visible to the viewer, without any need for culling like view-frustum, backface or occlusion culling. Further, we do not waste computation time by e.g. finding a reflection point in (u, v)-space, then projecting it to the viewplane and at last recognizing that it didn't contribute to the image anyway because it lies on a backface or is overwritten by some other pixel due to the z-buffer test (visibility-problem).

- In screen-space, we can easily judge the contribution of a mirror surface to the final image. By observing the pixel count the surface occupies in the frame-buffer, we are able to correctly assess its importance. We will see later that SSCR exploits exactly this information to perform an importance-driven point distribution between mirror surface partitions in the screen-space.

3.4.3 Fast screen-space root-finding algorithm

We will now present an efficient screen-space root-finding algorithm, which allows for fast mapping of scene points to their associated reflective mirror pixel. As discussed earlier, on planar and convex surfaces we know that there is at most one reflective pixel, while on concave surfaces there might also be more (if the camera point lies close to the focal point of the surface). We will therefore first formulate an algorithm which performs the mirror pixel search on regions of homogeneous curvature. Later we will extend our algorithm in order to render correct mirror reflections on surfaces of complex curvature.

Given the eye-point E, an arbitrary scene point Q and a raster image I of the perspective-projected scene containing a mirroring pixel region M^S of homogeneous curvature, we want to find as fast as possible the pixel $R^S \in M^S$ – if it exists – containing the projected reflection point R of the surface. We do this by a screen-space root finding in the mirror-space error function, continuously stepping towards R^S. To determine the direction of R^S in each step we use two points for mirror-space error comparison in order to estimate the next step. Since in the beginning we have no clue about the position of R^S, we start with two random seed points P_1 and P_2 within the pixel region.

At each pixel P_i, we estimate the 2D direction of the root by individually evaluating the root direction in 1D for both the x and the y dimension of the screen space (respectively mirror space, respectively the root line orientations which we assume to be aligned with them). A step in 2D is then performed by an individual x-step and y-step based upon the two mirror-space errors δ_x and δ_y, at certain step-sizes (see Figure 3.12(b)).

For each 1D direction i, we determine the direction of the root by comparing the signs and absolute values of the errors $\delta_i(P_1)$ and $\delta_i(P_2)$. At each step, P_2 is offset to a new position P_2' according to the step direction, and P_1 is set to the old position of P_2.

We know that we have overstepped the reflection point R if we get different signs for the errors $\delta_i(P_1)$ and $\delta_i(P_2)$ for in both the x and the y dimension.

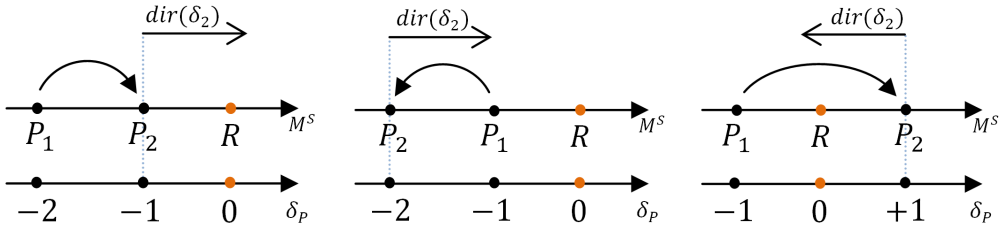
After a step from P_1 to P_2, in order to determine the next step direction in 1D we distinguish the following three simple cases based on the errors $\delta_i(P_1)$ and $\delta_i(P_2)$:

1. $sign(\delta_i(P_2)) = sign(\delta_i(P_1))$ and $|\delta_i(P_2)| < |\delta_i(P_1)|$:
   P_2 still on the same side of the root as P_1, but stepped closer to it.

2. $sign(\delta_i(P_2)) = sign(\delta_i(P_1))$ and $|\delta_i(P_2)| >= |\delta_i(P_1)|$:
   P_2 still on the same side of the root as P_1, but stepped away from it (or at least, didn't come closer).

3. $sign(\delta_i(P_2)) \neq sign(\delta_i(P_1))$:
   P_2 stepped onto the other side of the root (change of error sign), thus the root has to be located somewhere in-between (P_2) and (P_1).

Figure 3.16 illustrates these cases. If the sign of δ_i has not changed, and its absolute value was reduced, we proceed stepping in the current direction. If we have moved away from the root or passed it, we have to turn around, i.e. change the direction.



(a) approaching R (b) stepping away from R (c) passing R

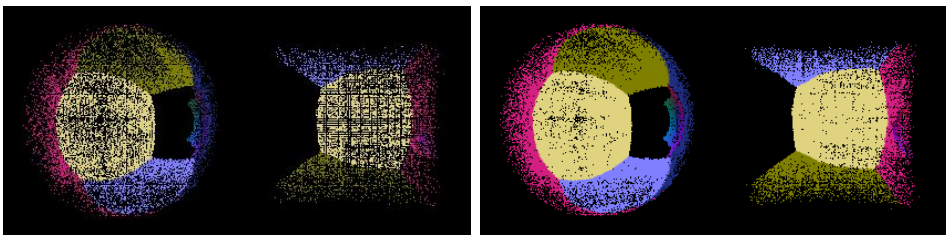
Fig. 3.16: Illustration of the three cases for determining the root direction $dir(\delta)$ after a step from P_1 to P_2. (The values for δ_P are chosen arbitrarily.)

For both the x and y dimension we use individual step sizes, which relate to the extension of the pixel region in that dimension. Therefore, a tall (big y) but narrow (small x) region starts with a bigger y step size and a lower x step

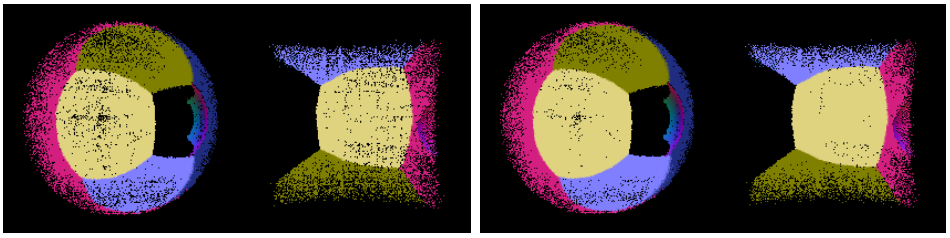
Algorithm 3.1 Screen-space root finding algorithm

$stepsize.xy \leftarrow diameter(M^S)/2$
$dir.xy \leftarrow (1, 1)$
$P_1 \leftarrow$<random pixel in M^S >
$P_2 \leftarrow$<random pixel in M^S >
while $P_1 \in M^S$ **and** $P_2 \in M^S$ **do**
  $\delta_1 \leftarrow mirrorSpaceError(P_1, Q, E)$
  $\delta_2 \leftarrow mirrorSpaceError(P_2, Q, E)$
  if $distance(P_1, P_2) < threshold$ **then**
    if $sign(\delta_1.x) \neq sign(\delta_2.x)$ **and** $sign(\delta_1.y) \neq sign(\delta_2.y)$ **then**
      $R.x \leftarrow interpolate(P_1.x, P_2.x, |\delta_1.x|/(|\delta_1.x| + |\delta_2.x|))$
      $R.y \leftarrow interpolate(P_1.y, P_2.y, |\delta_1.y|/(|\delta_1.y| + |\delta_2.y|))$
      if $frontFaced(\overrightarrow{RQ}, N_Q)$ **and** $frontFaced(\overrightarrow{PR}, N_R)$ **then**
        return R
      end if
    end if
    return n/a
  else
    if $sign(\delta_2.x) \neq sign(\delta_1.x)$ **or** $|\delta_2.x| \geq |\delta_1.x|$ **then**
      $dir.x \leftarrow -dir.x$
    end if
    if $sign(\delta_2.x) \neq sign(\delta_1.x)$ **then**
      $stepsize.x \leftarrow stepsize.x/2$
    end if
    if $sign(\delta_2.y) \neq sign(\delta_1.y)$ **or** $|\delta_2.y| \geq |\delta_1.y|$ **then**
      $dir.y \leftarrow -dir.y$
    end if
    if $sign(\delta_2.y) \neq sign(\delta_1.y)$ **then**
      $stepsize.y \leftarrow stepsize.y/2$
    end if
    $P_1 \leftarrow P_2$
    $P_2 \leftarrow P_2 + (dir.x * stepsize.x, dir.y * stepsize.y)$
  end if
end while

size. This way the step sizes can maintain a fast convergence in the longer dimension, while doing more careful steps in the small one. The initial step sizes are determined by a parameter of the algorithm, which describes them as a fraction of the region's extension in both directions. This fraction is a value between 0 and 0.5 and can be reduced for regions of difficult shape in



(a) 1 pixel threshold (b) 2 pixel threshold



(c) 3 pixel threshold (d) 4 pixel threshold

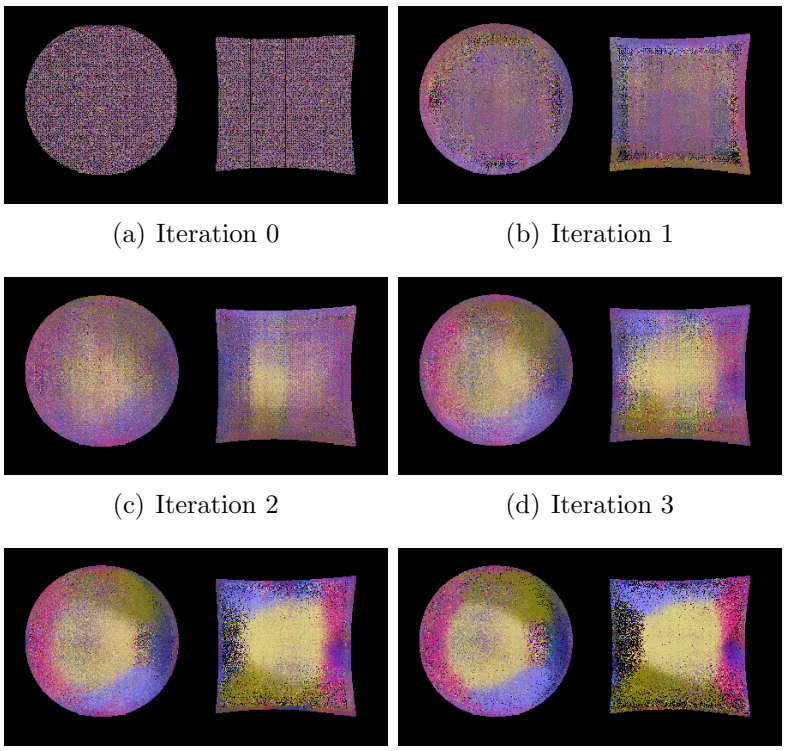
Fig. 3.17: Comparison of the mapping density for different thresholds in the demo scene in Figure 3.8. With increasing threshold, the success rate of the algorithm significantly increases, while allowing only an unrecognizable error. Note that for demonstration we chose a mirror pixel splat size of 1 in this scene.

the framebuffer, where more careful steps are intended. We proceed stepping by the same step size until we encounter the third of the three cases above, where we step over the reflection point and change direction. In this case, we reduce the step size, e.g. halve it. This procedure is performed until

- either P_1 and P_2 show different δ-signs in both x and y direction, and their distance reached some certain minimum threshold, or
- the pixel step size fell below 1 pixel

Note that even if there is a reflection point present in the pixel region, the second case where no reflection point is found can however occur due to possibly big alignment errors of the root lines, mapping errors from mirror-space to screen-space and the discreetness of the steps.

In the first case, we were able to narrow down the region in which R^S has to be located to a few pixels. In this case we interpolate the actual pixel position of R^S between P_1 and P_2 by the absolute values of their mirror-space deltas in both x and y direction. This approximation is eligible, since we deal with a small pixel region, and as we see later, we use point-splats to fill the region that remove this tiny error anyway. The smaller this threshold



(a) Iteration 0 (b) Iteration 1

(c) Iteration 2 (d) Iteration 3

(e) Iteration 4 (f) Iteration 5

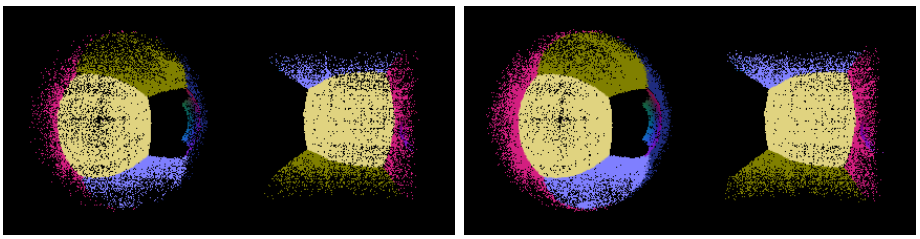
Fig. 3.18: Illustration of the pixel locations after several iteration steps in the demo scene in Figure 3.8. (Initial step size: 0.25x bounding-box size)

is chosen, the more exactly is the reflection mapping, but the bigger is the chance not to find R^S anymore, resulting in a sparser point mapping on the surface. Figure 3.17 compares the mapping density in a demo scene containing a spherical and parabolic mirror for different thresholds.

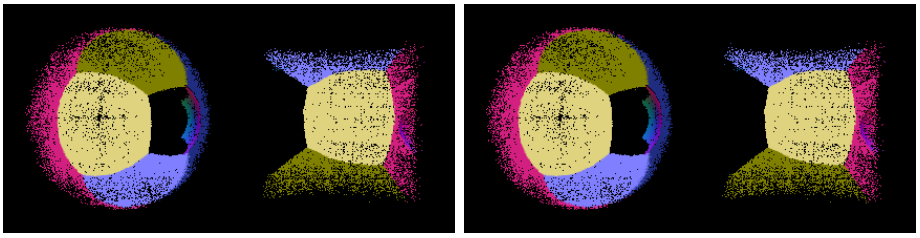
Concluding this, we can outline the final *screen-space root finding algorithm* that is shown in Algorithm 3.1.

Figure 3.18 illustrates the result of this algorithm after several steps. Starting with random pixel positions in iteration 0, the resulting image more and more converges towards the final sharp mirror image. Note that after a few iterations we can already identify the final mirror image.

Enhancing hit rate The objects shown in Figure 3.18 have simple shapes without holes and with a convenient relation between area and perimeter. In more common scenes, we can encounter homogeneous surfaces with projections of arbitrarily difficult shape. The pixel density of the mirror mapping can significantly decrease for narrow shapes or shapes with holes because of



(a) 0 attempts (b) 1 attempt



(c) 2 attempts (d) 3 attempts

Fig. 3.19: Enhancement of the mapping density in the demo scene in Figure 3.8. With increasing maximum number of seed- and step-attempts, the density at the shapes' border regions increases. (Initial step size factor: 0.25, Threshold: 4)

the higher risk of stepping out of the mirror region. Further, we can even lose points directly at the initial seeding step because we can only choose the seed position as a random point within the region's screen-space bounding box. Also, the chosen initial step size matters. Starting with a bigger step size enables us to reach our reflection point faster, but also increases the possibility of stepping out of the region. In order to obtain dense mirror mappings also for complex screen-space shapes, we can enhance our algorithm by the following two mean:

- At the beginning of our algorithm, we allow several attempts for seeding our initial points P_1 and P_2 before discarding them.
- At the end of each iteration, we allow several attempts of stepping to the next pixel P_2, each time decreasing the step size.

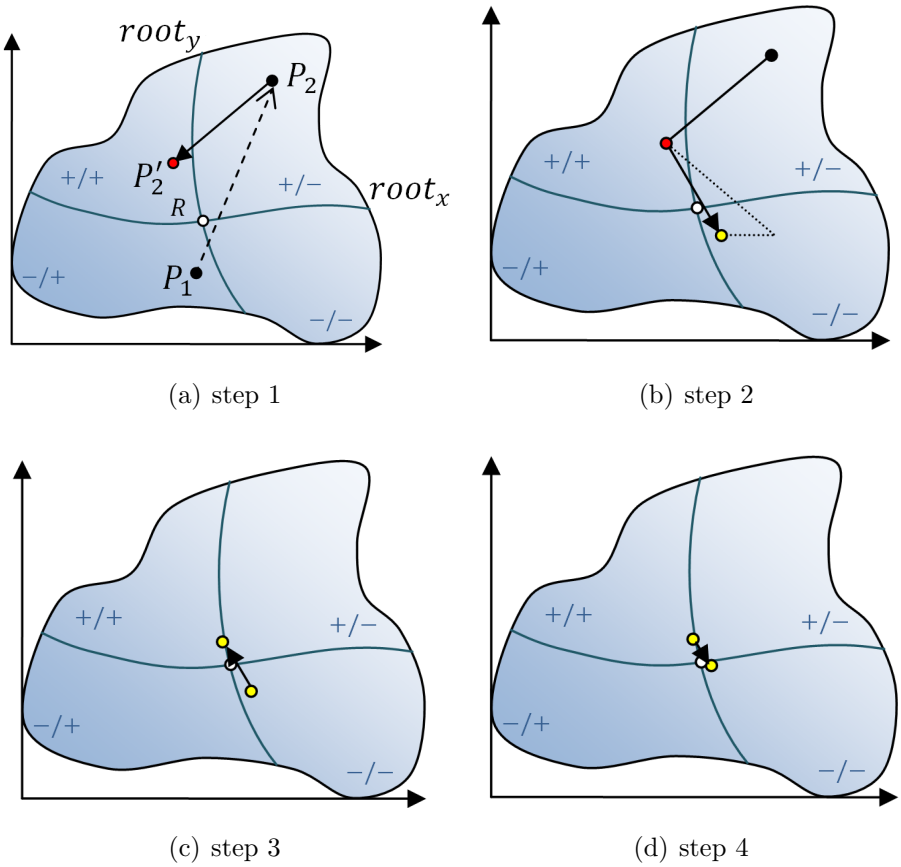
Although these enhancements increase the required time for the root finding, we obtain much better results in our images, as shown in Figure 3.19.

Discussion Our algorithm shows similarities to both the bracketing and the bisection method which are often used for root finding. As long as we observe the same error sign while stepping, we approach the root point in one

direction similar to bracketing. After encountering a change in the error sign, we turn around and proceed with reduced step size, since the root point R must lie somewhere in-between. Eventually the algorithm can converge after several of these turn-arounds, continuously narrowing down the region in which R is located, like a bisection algorithm does (see Figure 3.20).

On the other hand, we know that the root lines of our error function can project to arbitrarily curved lines, which are not necessarily aligned with the screen-space axes. Therefore, while narrowing down the borders, a conventional bisection could loose the root point due to exclusion. In contrast, our algorithm recovers by automatically applying an approaching bracketing-like step again. Due to these characteristics, we could call our 2D root-finding algorithm "interleaved bracketing and bisection".

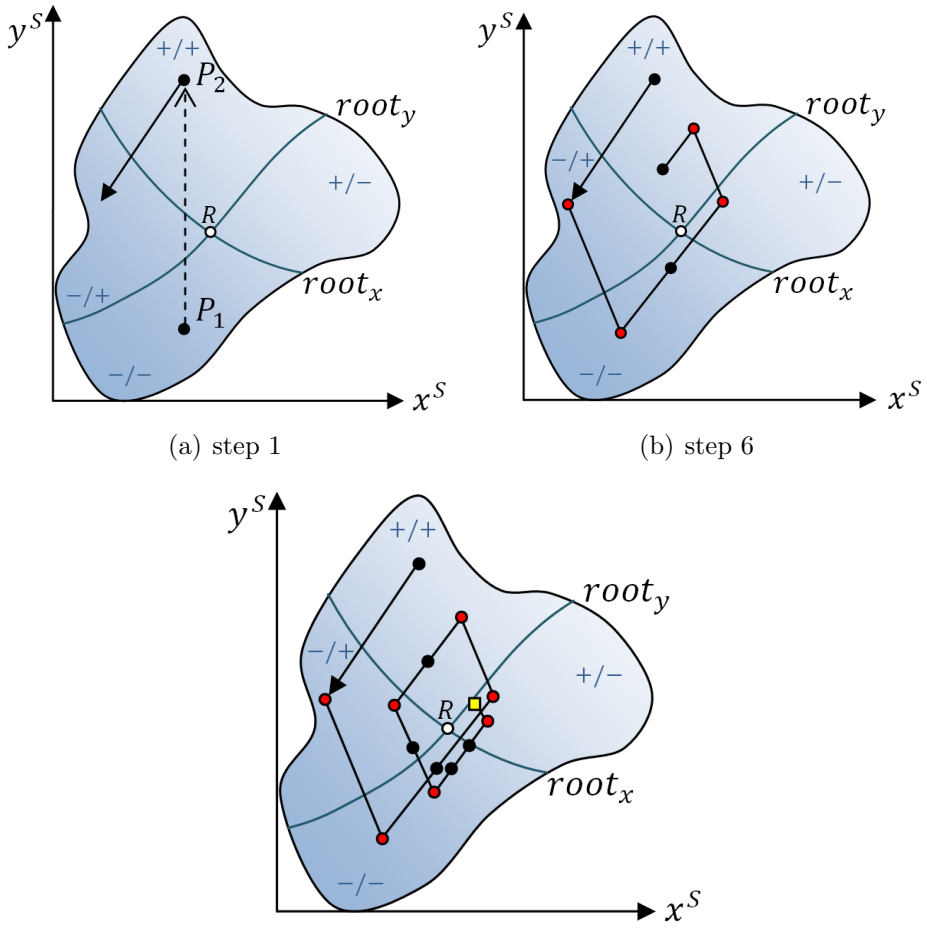
Figure 3.20 illustrates an example stepping procedure of our algorithm



(a) step 1 (b) step 2

(c) step 3 (d) step 4

Fig. 3.20: Example for a fast convergence of our algorithm on a 2D surface. (Initial step size: 0.25x region size)



(a) step 1

(b) step 6

(c) step 13

Fig. 3.21: Example for bad algorithm behavior, ending up in zero step size without finding the root point. (Initial step size: 0.3x region size)

on a 2D surface. The procedure contains the following steps:

(a) First, two points P_1 and P_2 are seeded in the surface. The first step is performed based on their relative error signs. Note that in this example the initial step size in x and y is 25% of the region extension.

(b) After the first step, we encounter a change of sign in the horizontal direction (red dot). Thus, the x-direction is inverted, and the x step size is halved. The step size and direction in y stays the same, as we have approached the root line in this dimension.

(c) After the second step, we encounter a change of sign in both dimensions (yellow dot). We turn around and reduce both step sizes.

(d) At the third step, we have to turn around again. Now, the region in which the root point is located is already sufficiently narrowed. Altogether, we found the reflection point after four steps.

In this example, we exploit the fact that the root lines are well aligned with the screen-space axes. This results in good estimations of the root direction.

However, our algorithm can also show bad convergence for different situations. Figure 3.21 gives an example for a case where the root lines are rotated approximately 45° in relation to the screen-space axes. We see that for start conditions similar to Figure 3.20, the algorithm needs many iterations. It continuously steps around the root point, but never finds it. In the previous example of fast convergence, the algorithm did a step that changed both error signs (yellow dots), which allowed for quickly narrowing down the root. In the case of bad convergence, we see that the trajectory of our stepping point always crosses only one root line, i.e. changing the direction only for one screen dimension (red dots). Since the root lines are aligned with the major step directions rather than the screen-space axes, a step never crosses the root in a way that changes both error signs. This way, the step size is continuously reduced to zero where the algorithm breaks (yellow box).

Such cases represent a waste of computation time. They need far more steps than the fast, successful cases, and do not contribute to the resulting image. Therefore, we clamp the 2D root finding procedure to a maximum step size count, which is a parameter of the algorithm.

3.5 Complex mirror geometry

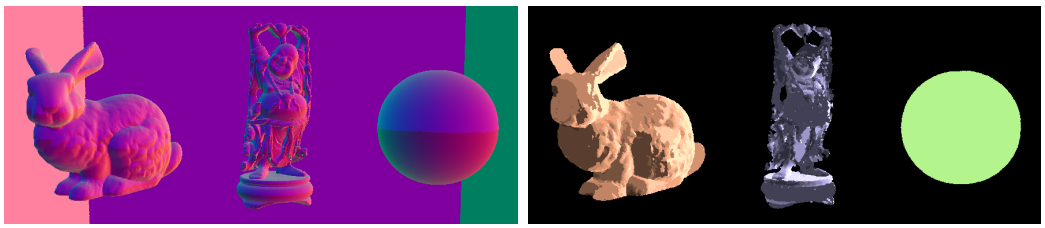
With the given algorithm, we can already render scenes containing several non-connected mirror surfaces of homogeneous curvature, as long as we know their screen-space bounding box for the calculation of the initial step sizes. A naive way to render the scene would be to test each scene point Q against each homogeneous pixel region M^S, considering that a point Q could be possibly reflected in each of the given region. However, in scenes that contain highly complex mirroring surfaces resulting in a huge number of homogeneously curved regions, computation time would explode. We therefore use a point distribution approach. In order to render the mirror reflections on the surfaces in the framebuffer, we pass every point through a vertex shader program exactly once. This vertex shader assigns it to a certain region, on which our root finding algorithm is executed. Thus, each scene point is tested against exactly one region. This is legitimated by the presupposition that we have on our point cloud scenes, namely that they contain a huge number of

points. We consider that in our application, common scenes contain at least several million points. If the whole framebuffer would be occupied by pixels that show mirroring objects, at common viewport sizes we would have only about one million pixels on which our points have to be distributed. In more realistic scenes, the relation between the number of scene points reflected in a mirror and the number of framebuffer pixels occupied by the mirror is even bigger.

So far, we have only discussed mirroring surfaces of homogeneous curvature. For these types of surfaces, the mirror-space function has a certain root, i.e there is at most one reflection point. As we have seen, on concave surfaces there can already be more than one, even if the surface curvature is homogeneous.

On more complex surfaces consisting of multiple patches of homogeneous curvature, the mirror-space error function plot shows a hilly surface, containing multiple valleys of different size containing a reflection point at their root. Applying our algorithm to such functions as is raises the problem that it cannot perform a fast root finding anymore. Choosing an arbitrary initial step size, we always face the risk that it is too big for the local valley our seed point lies next to. When performing a step from a given P_1 to a P_2 that passes several function maxima, the algorithm could jump around on the function surface uncontrollably. Although this procedure may hit some actual reflection points from time to time, this method would not be feasible in general scenes. Since the initial step depends on the size of the homogeneous region, we cannot choose a globally suitable step size anymore, since we can encounter different regions of different sizes. We thus would have to reduce the step size to the size of a pixel in order to ensure not to perform an invalid step. This however would throw us back to a slow algorithm again, that is not applicable for real-time rendering anymore. Thus, SSCR implements a solution to this problem that keeps its ability to perform a fast root finding by big step sizes.

In each frame, before applying the root finding pass on the scene points for reflection rendering, we perform a screen-space segmentation pass on the viewport image. Each segment is then labeled in order to create a buffer that contains information about the extent of each homogeneously curved mirror surface patch in the image. We call this buffer *labeled homogeneous curvature map* (LHC map). It contains an image of the viewport's mirror surfaces where complex surface regions are split into its patches of homogeneous curvature. Within those patches, to each of its containing pixels we assign the unique ID of its homogeneous region. Further, we provide a second buffer containing the screen-space bounding box coordinates for each of those homogeneous regions. Figure 3.22 shows a schematic illustration of such an



(a) scene normal map (b) LHC map

Fig. 3.22: Schematic illustration of an LHC map (right) that is extracted from the normal map of a scene (left). Each region in the LHC map represents a surface part with homogeneous curvature, and is given a unique ID that is assigned to each of its pixels. The distinct IDs are illustrated by a certain color. (Note that while the normal map is an actual screenshot from our application, the illustration of the LHC map is just artistic, as are its colors). Bunny and Buddha model courtesy of Stanford Computer Graphics Laboratory.

LHC map. It shows that complex surfaces like the bunny or the Buddha in the figure would be decomposed into a big number of homogeneous regions, while a simple sphere would result in only one region.

With the availability of the LHC map and its bounding boxes, we now can easily handle complex mirroring surfaces when applying our screen-space root-finding method. In the vertex shader for reflection calculation, each incoming vertex is assigned to one of the homogeneously curved regions in the scene, and seeded within this region based upon its bounding box information. Knowing the ID of the region a point was originally assigned to, we can now perform our screen-space root finding by stepping through the pixel region at appropriate step sizes. To test whether the region was left, after each step iteration we lookup the region ID of the new pixel in the LHC map and compare it to the original one.

The detailed segmentation and labeling steps that are executed in order to create this LHC map are described in the following.

3.5.1 Curvature map creation

First, we render a *curvature map*, which contains the information about local discontinuities in the curvature of the mirror surfaces associated with each framebuffer pixel. These discontinuities represent the border of a region of homogeneous curvature, and can be either of the following:

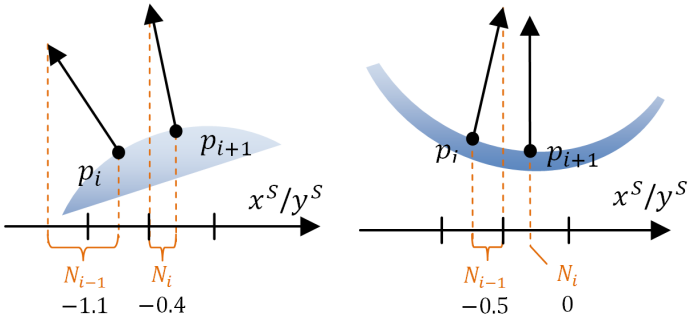
- change of curvature (inflection points)
- discontinuous depth step (one curved surface lying behind another)
- edge of mirror surface

The curvature information is extracted from a depth and a normal map of the viewport image, which are already available as an intermediate result of our conventional GI rendering pipeline. In the curvature map, each pixel stores only a qualitative curvature information that is represented by flags in the curvature map. These flags can indicate one of the three types, i.e. planar, convex or concave surface curvature. This qualitative curvature representation is sufficient for our further algorithm.

Both changes of curvature and depth steps are evaluated per pixel on the depth and the normal map. We test discontinuities in two directions: once in horizontal and once in vertical screen-space direction. For each pixel, we compare its linear depth and its surface normal with those of its previous (left or upper) pixel. Depth discontinuities are determined by a certain threshold parameter *depthStepThreshold*, which indicates the maximum depth difference for two pixels to be interpreted as parts of a continuous surface. Algorithm 3.2 illustrates the determination of a depth step between a current pixel's depth d and the depth of the previous pixel d'.

Based on the change of the normal in the respective directions (i.e. the normal's x or y component in view space), for each pixel we write its local curvature, i.e. positive (convex surface), negative (concave surface), or zero (planar surface), to the curvature map. Figure 3.23 illustrates the curvature calculation from the surface normals for the convex and the concave case.

Algorithm 3.3 shows the calculation of the both the x and the y curvature flags for the curvature map pixel c based on the i-th component (x or y) of



(a) $\Delta N_i > 0$ (b) $\Delta N_i < 0$

Fig. 3.23: Per-pixel determination of the curvature by the sign of $\Delta N_i = N_i - N_{i-1}$

Algorithm 3.2 Determination of depth discontinuities in direction i

if $abs(d - d') > depthStepThreshold$ **then**
    $setFlag(c, DEPTHSTEP_i)$
end if

Algorithm 3.3 Curvature extraction in direction i

if $abs(n_i - n'_i) < planarNormalTreshold$ **then**
    $setFlag(c, PLANAR_i)$
else if $n_i > n'_i$ **then**
    $setFlag(c, CONVEX_i)$
else
    $setFlag(c, CONCAVE_i)$
end if

the normal n of the pixel and the normal n' of its previous neighbor. The *planarNormalTreshold* parameter determines the tolerance for the difference of two normals in order to be interpreted as planar.

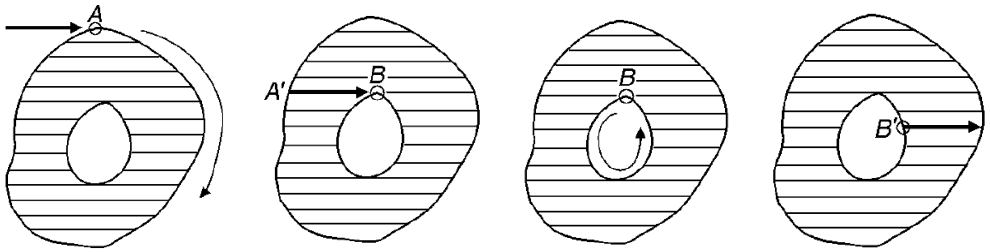
Thus, each pixel $P_{x,y}$ of the curvature map contains the following information:

- x curvature from $P_{x-1,y}$ to $P_{x,y}$ (negative, zero or positive)
- y curvature from $P_{x,y-1}$ to $P_{x,y}$ (negative, zero or positive)
- discontinuous depth step from $P_{x-1,y}$ to $P_{x,y}$ (true or false)
- discontinuous depth step from $P_{x,y-1}$ to $P_{x,y}$ (true or false)

With the information in the curvature map, we already gained a partition of the framebuffers's mirror surface regions by the depth differences. It is not yet segmented by its inflection points, i.e. we only have a map of per-pixel curvature signs. Segmentation by change of curvature is performed in the next step as part of the labeling algorithm.

3.5.2 Homogeneous curvature labeling

After creating the curvature map, we want to label each pixel with the unique ID of the homogeneous region it belongs to. In comparison to the common connected component labeling (CCL) task, our problem is a little bit more complex, since we do not just label independent disconnected regions in a binary image, but rather have to deal with neighboring components in the curvature map, and distinguish different regions by their different curvature.



(a) tracing external contour (b) encountering internal contour (c) tracing internal contour (d) labeling inner pixels

Fig. 3.24: The four major steps of the contour-tracing based CCL algorithm. If the scan encounters an external or internal contour of a region, the algorithm starts tracing the contour and labeling its pixels. This way, the algorithm can always distinguish between inside blob pixels and pixels belonging to holes in the blob when labeling. Image courtesy of Chang et al. [CCL04].

Generally, several parallel and sequential CCL approaches have been developed. SSCR implements an adapted version of the contour-tracing based CCL algorithm introduced in [CCL04]. This is a sequential algorithm with a time complexity of $O(n)$. Starting with the top-most scanline, it runs through each scanline of a binary image. When hitting a foreground pixel (i.e. the external contour of a blob), it assigns the next label to that pixel and starts tracing that contour, marking the contour pixels with the same label. If the scan encounters a hole in the blob, its internal contour is traced and labeled in a similar way. This way, when processing consecutive scanlines, it can easily fill inner pixels of the blob by assigning the label of the left neighbor, starting with the first labeled contour pixel it encounters on a scanline and stopping at the next one. Figure 3.24 illustrates the major steps of this algorithm.

By its nature, the CCL algorithm can assign different labels only to regions that are not connected. However, in our framebuffer, we want to label different regions of homogeneous curvature, which are connected to one big mirror surface. Thus, we have to slightly adapt the contour-tracing based CCL algorithm by [CCL04], and perform some additional processing on our curvature map in order to be able to apply the algorithm to it. These steps are described in the following.

1) Smoothing curvature Before we start, we need to smooth the curvature map. Due to its discrete nature, the curvature map always shows discontinuous curvature information along a scanline, especially when operating on

a framebuffer containing point splats. For example, a continuously convex surface can contain several planar pixels among the convex ones. This is due to the rendered point splats that cover several pixels in the G-Buffer, and to the limited accuracy of the float datatype by which the normals are encoded that determine the curvature. For such a convex surface, the curvature signs of a possible curvature map scanline thus could be

0++0+00++−0++−+0 [Input]

The smoothing step simply runs through each scanline horizontally and vertically, adjusting each pixel with zero curvature to the curvature of the previous pixel. Since we also have to consider that there could be no curved pixels at all (planar mirrors), this adaption for zero-pixels is only performed if the previous pixel already has a curvature. For the scanline above, our smoothing step would produce the following result:

0+++++++++−−++−++ [Smoothed]

After this smoothing, we actually have 3 different regions left in this scanline. Although this way to count disregards the planar pixel at the beginning of the scanline, this does not matter as we will see later. Note that there are however a few issues with this method. For example, it could overwrite an actual planar region neighboring a convex or concave one. For convex neighbors, this doesn't matter since the root direction in the mirror-space error function is the same as for planar mirrors anyway. Otherwise, for concave neighbors this could result in wrong labeling. However, the smoothing step mainly has to prepare the noisy, aliased screen-space curvature information in the curvature buffer for labeling.

2) Suppressing contour pixels In order to apply the CCL algorithm to our framebuffer containing *connected* regions which should be labeled differently, we apply a trick: We temporarily mark all pixels representing border pixels between different regions as "suppressed" by a flag in the curvature map. Based on this flag, the CCL algorithm can determine the border of a region, i.e. it is able to recognize two regions as disconnectd. This also implies that these suppressed pixels are not labeled in the following pass. Therefore, we append an additional pass after labeling, which closes these holes. In order to suppress the contour pixels, we simply run through each pixel of the image and mark a given pixel as suppressed if in x or y direction it has

a) a depth-step in relation to the previous pixel,

b) a different curvature sign than the previous pixel.

After this pass, we have a curvature map that provides a distinct information bit for each pixel that represents a border between different homogeneously curved surface patches, be it due to depth steps or inflection points. The scanline from the previous example would now contain suppression flags S at the following pixels:

`S++++++++S-S+S++` [Suppressed]

3) Adapted contour-tracing based labeling In this step, we apply an adapted version of the contour-tracing based CCL algorithm to the smoothed curvature map containing suppressed contour pixels. This algorithm assigns to each surface pixel a unique ID of the homogeneous region it belongs to. The conventional CCL algorithm takes a binary image as input, and traces the contour of a blob by differentiating between foreground (blob) pixels (1 bits) and background pixels (0 bits). In principle, our adapted CCL algorithm works the same. The only difference to our algorithm is that we do not operate on a binary image where we recognize blob borders as foreground pixels adjoining a background pixel, but rather process our curvature map and recognize blob borders as curvature pixels adjoining either a background pixel (no mirror surface at all) or a suppressed contour pixel that was marked in the previous step (depth-step or inflection pixel). Therefore, suppressed pixels are interpreted as background pixels in this step, thus they are not labeled, though they too belong to the surface regions. After this step, our example scanline would show the following values:

`S11111111S2S3S44` [Labeled]

Note that besides the labeled curvature map, this step produces two other outputs:

- a buffer containing the bounding box of each homogeneous region, and
- a buffer containing the pixel count of each region.

This information can simply be collected while executing the sequential CCL algorithm: For each pixel that is labeled with a given label L, we increase the pixel counter for L in a global counter array. Further, we test the screen-space coordinates of L against the respective current bounding box and update the bounding box, if L lies outside. We need the bounding box information for seeding our points within the regions in screen space when performing our root finding algorithm. The reason for tracking the pixel count is explained in Section 3.6.

4) Labeling of suppressed border pixels After the common labeling process is finished, we still have holes in our labeled map where the suppressed contour pixels are located. In order to fill these holes, we apply a final pass, which simply assigns to each suppressed pixel the label of its – depending on availability – rightward or lower mirror pixel. The labeled scanline from our example now would finally look like the following:

`1111111112233444` [Border-filled]

Note that having a single planar pixel after the first step (smoothing) at the beginning of the scanline does not matter, since it is assigned to its rightward region's label anyway.

3.6 Mirror pixel density

As we have already seen, rendering each scene point to a single pixel when performing the mirror pass does not produce completely dense mirror images in the framebuffer. Generally, the pixel density achieved by a mirror pass depends on the following factors:

- Number of points in the scene
- Number and size of homogeneous surface regions in screen space
- Surface complexity (problematic concave cases)
- Shape complexity of the projected regions
- Algorithm parameters (initial step size, threshold, seed and step attempts)

Pixel density is also affected by the fact that we test each scene point only against one LHC region, as stated earlier. Further, in complex mirror surfaces we can encounter difficult LHC region shapes that have a high rate of lost pixels, i.e. very narrow regions or regions containing holes. Figure 3.25 shows an example for the mirror pixel density in a non-trivial scene, rendered using weighted point distribution, splat size 1 and no pull-push closure. In this example, we face several difficulty factors. First, the relative pixel space occupied by mirror surfaces is relatively high. Next, the Stanford Bunny represents a complex surface that is partitioned into about 200 regions in this scene. Finally, the torus (without the sphere on it) is recognized as one single region. However, this region contains a big hole in screen-space,

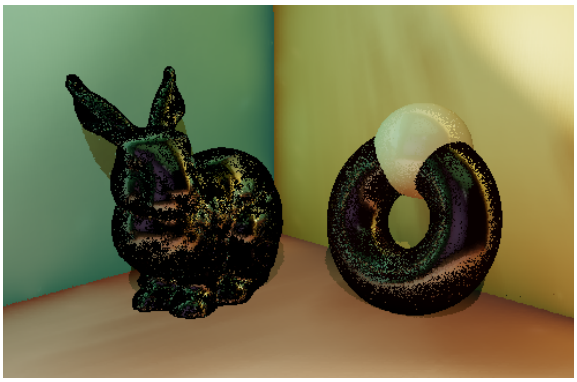


Fig. 3.25: Mirror pixel density in a general demo scene containing several mirrors of different complexity. (Bunny model courtesy of Stanford Computer Graphics Laboratory)

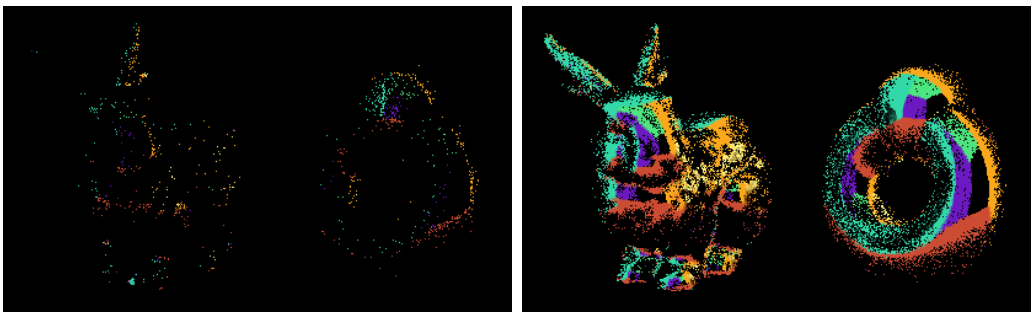
meaning that seeding and stepping on this region is rather prone to pixel losses.

Maintaining an overall high mirror pixel density even in complex scenes is rather difficult. Our SSCR algorithm takes the following measure to improve the pixel density:

1) Weighted point distribution We already mentioned that testing each scene point against each LHC region is not an option when trying to maintain an adequate performance. Therefore, each pixel is assigned to only one region. The easiest way to do this is using the running ID of the input point in a vertex program and perform an equal distribution to the LHC region IDs by a simple modulo operation. However, this method would assign the same number of points to very small regions, where many may be dispensable, as to very large regions, where more points could be required, Thus, our SSCR algorithm performs a stochastic point distribution based on a weight that correlates with the number of pixels in a region.

This way, we are able to assign most of the points to those regions, where they are actually needed. Figure 3.26 compares both equal and weighted point distribution in our complex demo scene. The figure points out the benefit.

2) Mirror point splat size The most obvious approach to achieve dense images is using point splats. When finding a reflective pixel R^S in the framebuffer for a given scene point Q, we do not only shade R^S but rather render a point splat of certain size that is located at R^S. Choosing the right point size is strongly case-dependent. Reflected objects close to a mirror surface may

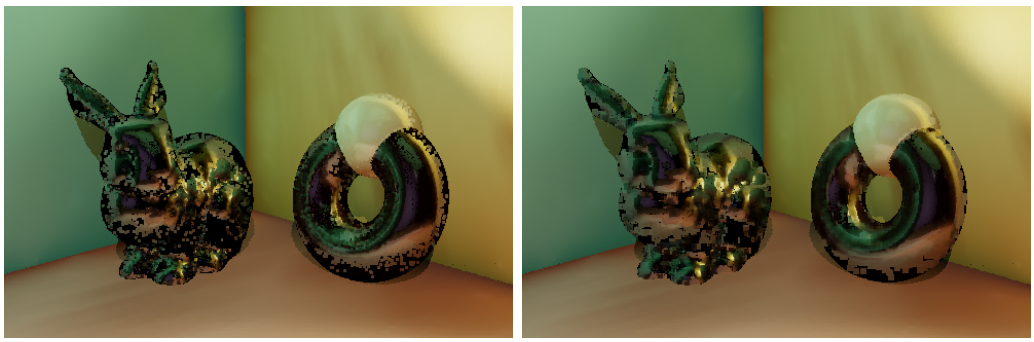


(a) equal point distribution (b) weighted point distribution

Fig. 3.26: Comparison of equal and weighted point distribution for the mirror buffer of the demo scene in Figure 3.25.

require bigger splat sizes, while using too big screen-space splats for more distant objects would lead to too coarse mappings.

Our algorithm thus uses a linear falloff between 1 at some maximum distance visible to the mirror (mirror z-far), and a certain maximum splat size (distance zero). This is more convenient to gain both high density for close objects and high detail mappings for far objects. Figure 3.27 compares mirror mappings of different splat sizes. We see that with increasing splat size, the mapping gains density, but also looses its sharpness. Note that depending on what maximum visible mirror distance is chosen, some mirrored point clouds could still look somehow coarse due to the size warping of mirrored images on curved reflectors. This issue is addressed later in Section 3.7.

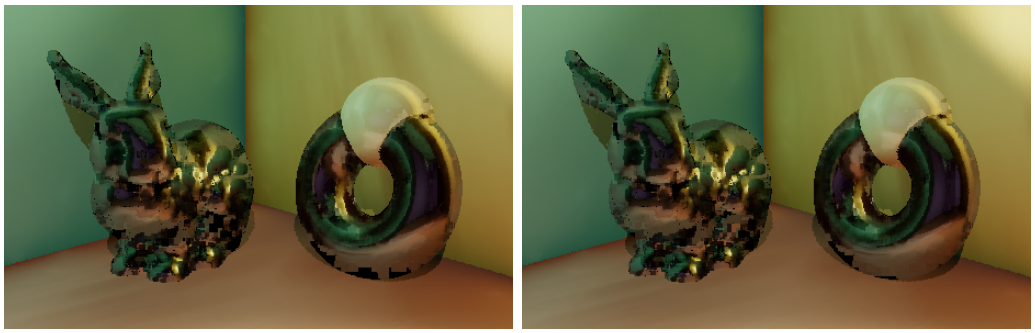


(a) 3 pixel max. splat size (b) 6 pixel max. splat size

Fig. 3.27: Demo scene of Figure 3.25 rendered with different maximum splat sizes. With increasing splat size, the mirror image starts looking smudgy.

3) Pull-push closing In order to fill remaining holes in the mirror image, we apply a pull-push pass to our resulting mirror buffer [SDG+98][MKC08]. In the pull-pass, similar to mip-mapping, the mirror buffer is down-sampled by a factor of 2 over several iterations, averaging pixel information. In a consecutive push-pass, the averaged pixel values are passed down the resulting image pyramid, filling pixels that contain no information (i.e. holes in the image). Although this method provides a fast and efficient way to fill also bigger holes, applying too many iterations can quickly result in visible artifacts or in pixel fillings where none are intended at all. Thus, depending on the scene our algorithm only applies zero up to four iterations (meaning that a given mirror pixel can already influence up to a 16 pixel wide region).

Figure 3.28 shows our demo scene after different numbers of pull-push iterations. With increasing iteration count, the holes at the bottom of the torus are filled, which represent missing mirror pixels. Simultaneously, the pull-push pass wrongly fills the open side of the enclosing Cornell box that is mirrored at the right end of the torus.



(a) 2 pull-push iterations (b) 4 pull-push iterations

Fig. 3.28: Comparison of the appearance of the demo scene in Figure 3.27(b) (6 pixels max. splat size) after 2 (left) and 4 (right) pull-push iterations.

3.7 Mirror visibility

So far, our algorithm is able to find the reflective pixel for a given scene point and shade the pixel with the attributes of that point. Until now we have not considered visibility for the reflecting surface point. Simply splatting all scene points onto a surface produces a mirror image on which occlusion depends on the splatting order, not on the depth. We therefore have to account for visibility in the mirror image.

The simplest way to handle visibility is to use the z-buffer and enable depth testing: We can choose a maximum visible depth, i.e. a *mirror z-far* value. Then, when transforming a vertex Q to its viewport location that corresponds to the reflection point R, we can easily calculate the distance between R and Q, normalize it by our chosen mirror z-far and use this normalized value as depth value for the current vertex.

Although this *nearest-point* approach is straight forward and produces correct mirror visibility, it can produce bad mirror image quality due the following two problems:

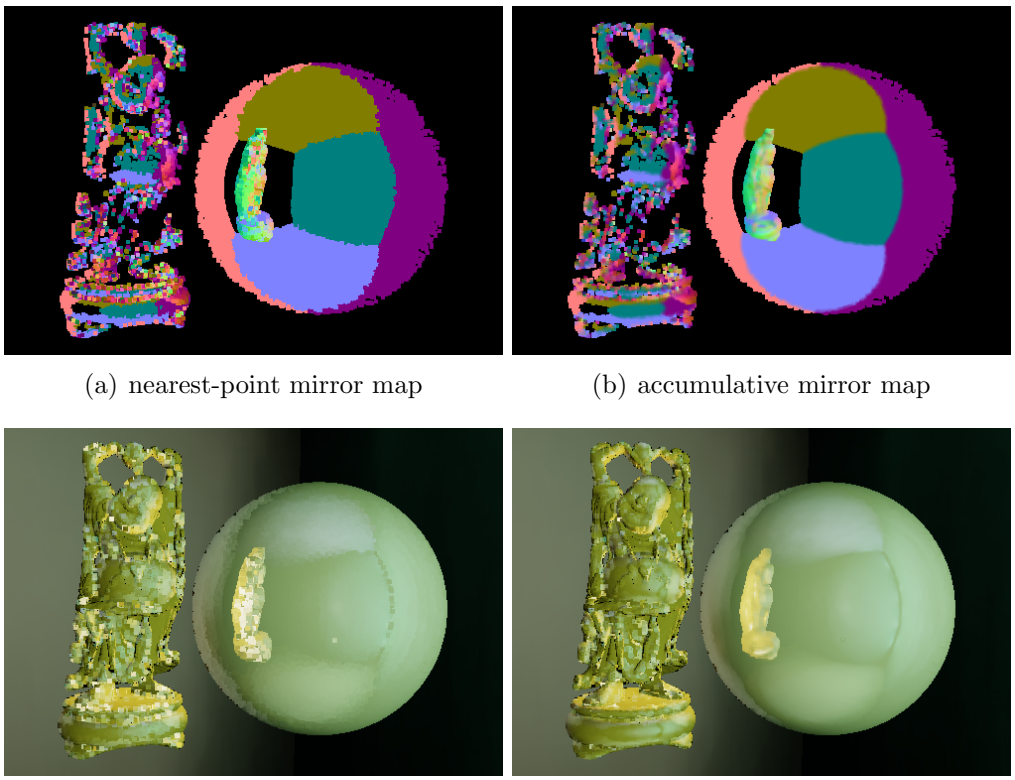
1. Using nearest-point, a mirror shaded pixel of the framebuffer can only reflect the information of one single point in the scene. Considering curved mirror surfaces, we often encounter a warped mapping of their environment, i.e enlarged or shrunk mirror images. In such cases, a single mirror pixel can contain the information of reflected light of a wider section of the scene. Thus, shading a pixel with the point splat fragment of the nearest scene point can introduce aliasing artifacts (e.g. color-flipping while moving the camera or the object).

2. Because of the overlapping point splats on the mirror, mapping detailed geometry or corners can result in box artifacts. Figure 3.29 shows an example.

In order to reduce artifacts and avoid visible aliasing effects in our mirror images, our algorithm accumulates the reflections of several scene points in each pixel. This way, mapping point splats which carry their reflection information to neighboring pixels is not prone to aliasing artifacts anymore, since it introduces a smoothing over the reflections of neighboring pixels. Further, image quality of mapped detailed geometry or corners can be improved. Detailed geometry may still appear vague, but this way they show a smoothed shape rather than a bunch of box splats. For example, the appearance of mapped corners can also be significantly improved, as the accumulation removes step-artifacts as shown in Figure 3.29.

In order to render nice accumulated mirror mappings, using the z-buffer is not convenient anymore. We have to introduce a two-pass algorithm [RPZ02] that is capable of accumulating points while still maintaining visibility. In a consecutive pass, we average the accumulated point attributes to obtain a final smoothed mirror image.

In the first pass, the mirror reflected points are rendered to a depth buffer, using the nearest-point method with the z-buffer described above. The resulting buffer is an image that contains the distance information of the nearest reflected point for each of its pixels.



(a) nearest-point mirror map (b) accumulative mirror map

(c) nearest-point mapping in GI (d) accumulative mapping in GI

Fig. 3.29: Comparison between nearest-point mirror mapping (left) and accumulative mapping (right). The upper row shows the points as they are splatted into the mirror map. The lower row shows the result of each method when applied on the GI scene. Due to the accumulation, box artifacts that are inherent in point splats are automatically blurred to smooth surfaces. Note that the accumulation mapping also produces much better mirror images of the room's corners.

In the second pass, the mirror image is rendered again, this time activating one-plus-one blending for the target pixels. In this pass, the output mirror buffer is shaded by the actual point attributes of the scene points. Simultaneously, we create a counter buffer that simply sums up the splat fragments that are shaded to each pixel. The visibility test for a given scene point Q is now performed by comparing the mirror depth value of Q to the value stored in the depth map that was rendered in the first pass. Similar to shadow-mapping [Wil78], we use a certain shadow-offset, by which we can define a depth range of pixels visible behind the nearest point, i.e. the number of points that are accepted for accumulation.

The third pass then simply normalizes the value of each pixel in the re-

sulting framebuffer. It looks up both the pixel's accumulated point attributes in the mirror buffer and its number of accumulated fragments in the counter buffer. The result is an output image containing the point attribute values normalized by the counter.

3.8 Multiple mirror bounces

In order to calculate first-order mirror reflections, we apply the methods developed in the previous sections only once. Based upon a normal map of the scene that is provided by our given GI framework in a Camera G-Buffer, we create an LHC map which is used to distribute the points in the scene between the visible homogeneous mirror surface regions. These scene points are splatted onto the mirror regions and stored in a new buffer, the *Mirror G-Buffer*, which contains various information about the scene point, like position, normal, color, etc. Based upon this Mirror G-Buffer, we are able to perform a GI shading for the mirrored scene images similar to the normal scene geometry. The result is blended over the original, diffuse-specular image.

In order to render higher-order mirror reflections, we can use the 1^{st}-order Mirror G-Buffer, and execute the complete SSCR algorithm once again. This time, we use the normal map and the linear depth map as input for the algorithm, both of which are part of the 1st-order Mirror G-Buffer. Curvature is now extracted only from mirroring surfaces which are already mirrored in a visible mirror surface. LHC map creation and Mirror G-Buffer pass are performed on this – mostly smaller – 2^{nd}-order mirror pixel regions. The GI shaded result is then once again blended over the current output image.

This process can be executed iteratively, each time adding a further mirror bounce. However, performance and interactivity are expected to be immensely affected after a few iterations. Considering that with each mirror bounce the number of visible n-th order mirror pixels is reduced in the most cases, it should be sufficient to pass only a subset of the points used in the previous bounce through the Mirror G-Buffer shader that performs the root finding on each point. Figure 3.30 illustrates the rendering pipeline for multiple mirror bounces. Note that although the algorithm is designed to support multiple mirror bounces, they are not yet realized in our application.

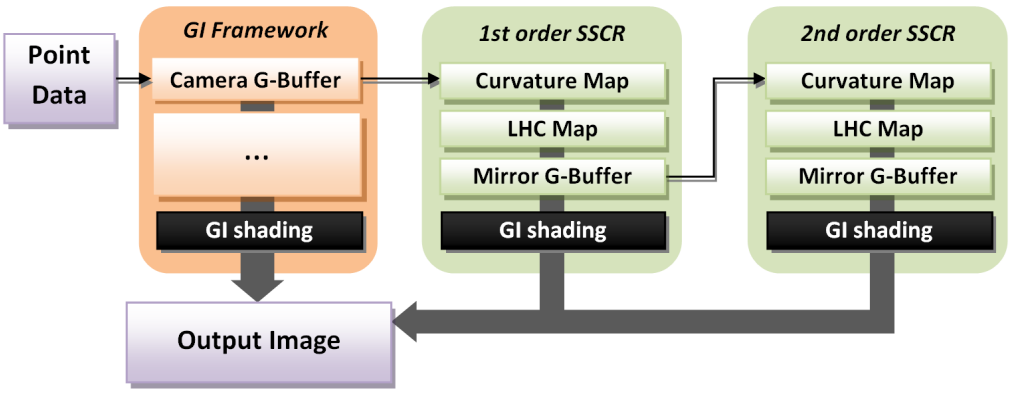


Fig. 3.30: Schema of the GI rendering pipeline incorporating SSCR rendering two mirror bounces. The data from the Camera G-Buffer is used as input for the first mirror bounce. Each subsequent mirror bounce can be performed based upon the Mirror G-Buffer from the previous bounce.

Chapter 4

Implementation

This chapter takes a closer look at the implementation of the SSCR algorithm in our point cloud renderer. It revisits the points discussed in the previous chapter, and describes the steps of the SSCR rendering pipeline in more detail.

Our algorithm is implemented in C++, using OpenGL 2.1 as graphics API and CG shader profile 4 for our shader programs.

4.1 Current GI algorithm overview

Lets first have a look at the current global illumination (GI) algorithm. The theoretical background to our current GI renderer was already given in Section 2.4.1. Figure 6 illustrates the rendering pipeline that the GI algorithm performs to render a frame.

In the first two passes, the points in the scene are rendered to a *Camera G-Buffer* and a *Light G-Buffer*. The Camera G-Buffer shows the scene from the perspective of the camera, the Light G-Buffer from the perspective of the (spot) light source. Both G-Buffers store various information about the surface point associated with each of its pixels, like linear depth, surface normal, diffuse color, etc. A number of virtual point lights (VPLs) are distributed in the Light G-Buffer, i.e. over the part of the scene visible to the light source. In the next step, the points in the scene are rendered again in order to create the combined ISM buffer for the VPLs, which contains their visibility information necessary for shading. With the VPLs seeded and their visibility given by the ISM buffer, we then perform an interleaved indirect illumination shading of the G-Buffer pixels in screen space. If multiple light bounces are required, the latter three steps are repeated: The VPLs are redistributed based upon their current location and visibility, the ISM buffer is recreated based upon the new VPL locations, and interleaved shading is performed again, blending the illumination of the new light bounce iteration with the previous one (accumulative shading).

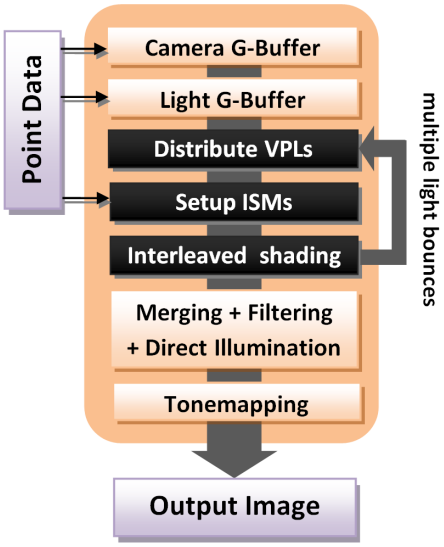


Fig. 4.1: Overview of the GI rendering pipeline in our point cloud renderer.

When shading of indirect light bounces is done, the split interleaved buffer is merged to an image of full viewport size again. This merged image is then filtered in order to obtain a smooth indirect illumination of the image throughout the whole image. After adding direct illumination with shadow mapping, the output image is tonemapped and then written to the back-buffer.

4.2 SSCR algorithm overview

Figure 4.2 gives an overview over the steps of our SSCR rendering pipeline. Based upon the Camera G-Buffer provided by our GI framework, we perform a curvature extraction pass. The result is a qualitative curvature map of the scene, which is used in the next step for the setup of an Labeled Homogeneous Curvature Map (LHC map). This map partitions the mirror surfaces in the G-Buffer into several regions of homogeneous curvature. Along with this map, we produce the bounding box coordinates and region-size based weights for the LHC regions. Based on this data we can perform a weighted distribution of the scene points to the LHC regions in order to create a Mirror G-Buffer (MGB). This MGB is created in three passes: a depth pass, a visibility pass and an averaging pass. On the resulting MGB, we perform a pull-push closing procedure to fill missing mirror pixel values with the averaged G-Buffer data of its neighboring pixels. Finally, we perform a GI shading of the mirrored surface points in the MGB, similar to the way it is

done for the remaining, diffuse and low specular part of the scene.

In the following, we will discuss the implementation-specific details of each of those steps.

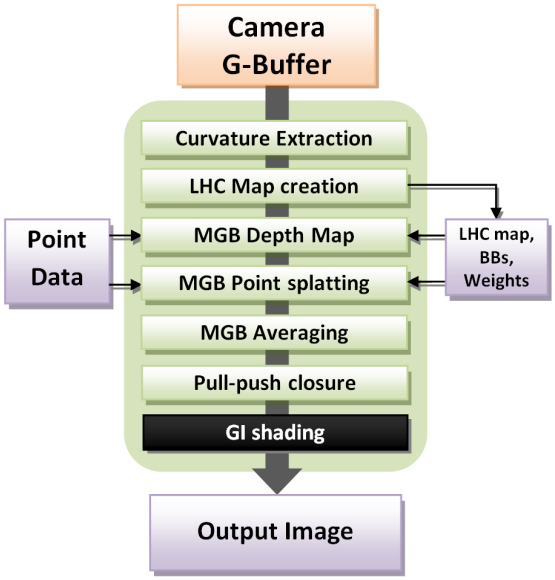


Fig. 4.2: Overview of the SSCR rendering pipeline.

4.3 Curvature extraction

Input buffers:	• Camera G-Buffer [normal, specular intensity, shininess] map • Camera G-Buffer [linear depth] map
Output buffers:	• Curvature map (1 channel byte)
Parameters:	*planarNormalTreshold, depthStepThreshold*

In the first step, we create a *curvature map* from the given Camera G-Buffer. This curvature map is needed in the consecutive LHC labeling step, which segments the viewport image and labels its LHC regions.

Curvature extraction is performed in one rendering pass. It renders a fullscreen quad drawing to a 1-byte single component target texture. Its input buffers are the normal map and the linear depth map of the Camera G-Buffer, since those two contain the required world-space normal and position used for continuous curvature determination. The fragment shader processes each

pixel of the output buffer, storing the information of curvature continuity in one byte per pixel. For this it uses the following byte setup (Figure 4.3):

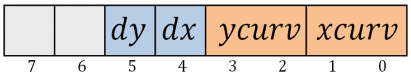


Fig. 4.3: Byte setup of the curvature map. Bits 6 and 7 are not used yet.

These curvature map entries are only rendered for pixels that represent a mirroring surface region in screen space. The rest of the buffer remains zero, as cleared before rendering. This way, the curvature map already provides a viewport segmentation, which is used later in the labeling pass, showing only mirroring pixels. Pixels representing a mirroring surface are identified by looking up the SI (specular intensity, i.e. shininess) information stored in the *Camera G-Buffer [normal, SI, SP]* texture. The bit flags in the above byte setup have the following meaning:

$xcurv$:	screen-space x curvature
$ycurv$:	screen-space y curvature
dx:	screen-space x depth step
dy:	screen-space y depth step

The $xcurv$ and $ycurv$ flags, each occupying two bits, indicate the change of the x respectively y-component of the view-space normal associated with a G-Buffer pixel from the previous to the current pixel, in horizontal respectively vertical scanline direction. It stores, whether there is a positive curvature, a negative curvature or no curvature present relative to the the previous pixel (i.e. the pixel left or top of the current pixel). This is simply determined by looking at the sign of the difference between the neighboring pixel's x or y components. Figure 3.23 in the previous chapter illustrates this concept. For both $xcurv$ and $ycurv$, curvature is encoded in the follwing way:

bits	normal-$\{x\|y\}$ change	curvature	surface type
00	-	-	-
01	descending	positive	concave
10	ascending	negative	convex
11	invariant	zero	planar

The dx and dy bits indicate whether there was a depth step from the previous pixel to the current one, i.e. whether there is a crack in the continuity

of the surface at this location. We use the *depthStepThreshold* parameter to determine the maximum depth two neighboring pixels may have in order to be interpreted as continuous surface.

When comparing two neighbored pixel's normals for equality, we have to be aware of float precision issues. Further, since our G-Buffer image is rendered by box splatting our point clouds, overlapping splats can lead to noisy normal maps showing a continuous change in curvature along a scanline, which it shouldn't. We therefore use the *planarNormalTreshold* parameter, that represents a maximum epsilon tolerance of the difference between two normal's x or y components, within which the normals are still interpreted as equal, i.e. interpreted as planar surface.

4.4 Homogeneous region labeling

Input buffers:	• Curvature map
Output buffers:	• LHC map (1 channel int32)
	• LHC bounding box map (4 channel float32)
	• sorted region map (1 channel int32)
	• sorted accum. weights map (1 channel float)

In this step, we create the *LHC map*, a buffer containing a labeled image of the mirroring surfaces, segmented into regions of homogeneous curvature. This is performed based on the curvature map that was created in the previous pass.

In Section 3.5.2, we have already presented our method and necessary steps for LHC map creation. We perform an adapted version of a contour-tracing based CCL algorithm [CCL04], which runs at $O(n)$ time complexity, but therefore is not parallelizable since it performs sequential steps on the input image. Therefore, we have to perform this task on the CPU.

In the first step, the whole curvature map texture is transfered from the GPU to the system memory. In order to reduce the data load that has to be copied, we have limited the pixel format of the curvature map to its minimum, i.e. 1-channel byte. This single byte stores all information needed for labeling. After labeling is done, the resulting LHC map has to be loaded back to the GPU by a new texture initialization. Since with one byte, our LHC map could only distinguish between 255 different regions, we use a 1-channel int32 pixel format for our LHC map. However, the fact that we have to copy the whole curvature and the LHC map between GPU and system RAM each frame does not represent a performance bottleneck in our

application (see Chapter 5). In our test scenes, the whole task (including data transfer) is performed in approximately 10 ms.

4.4.1 LHC map creation steps

We perform the following four steps for LHC map creation (see Section 3.5.2 for a detailed description):

1. Smoothing curvature
2. Suppressing contour pixels
3. Adapted contour-tracing based labeling
4. Labeling of suppressed contour pixels

Steps 1 and 2 operate directly on the curvature map that was copied to system RAM. They prepare the input buffer for the actual labeling process in step 3, which writes the labels to a new output buffer. Finally, step 4 post-processes the output buffer to obtain connected LHC regions.

1) Smoothing curvature The smoothing step directly alters the curvature information stored in the curvature map, according to the description in Section 3.5.2. We process each scanline and each column of the curvature map buffer, i.e. once in horizontal direction (smoothing $xcurv$) and once in vertical direction (smoothing $ycurv$). In each direction, each planar pixel that directly follows a curved one, retrieves the curvature information of that neighbor. Note that since this is a propagating process, this step already has to be performed on the CPU.

2) Suppressing contour pixels In the next step, contour pixels in the curvature map are marked as suppressed. Contour pixels indicate a depth step or a change of curvature and thus a border between homogeneous regions in x respectively y direction. Later in the labeling step, suppressed pixels are treated like background pixels, allowing for correct distinction of different regions and thus for correct labeling. Contour pixels are marked by setting a flag within the curvature storage byte of a pixel of the curvature map, where 2 bits are still unused. This is the S-flag shown in the extended byte setup in Figure 4.4.

3) Adapted contour-tracing based labeling In this step, we process each scanline of the prepared curvature map according to the contour-tracing based CCL algorithm of [CCL04], writing pixel labels in the output LHC buffer. Basically, our adapted version works the same as the original algorithm, showing solely the following three differences:

- We do not distinguish background pixels from foreground pixels in the original way. When tracing the contour of a region, a background pixel to that region is either an empty pixel (no mirror region at all) or a suppressed mirror pixel (looking at the *Supressed*-flag).
- The original algorithm performs an 8-connected contour tracing, i.e. originating from some pixel, it can follow the contour by stepping to each of its 8 surrounding pixels. In contrast to that, we perform a 4-connected contour tracing, since this method is more suited to our curvature map. This is due to the fact that the region borders, which are formed by the suppressed contour pixels, provide an 8-connected contour. Thus, performing an 8-connected contour tracing would step through these borders and unite different homogeneous regions. Note that this way, outlying pixels that connect with a region only by one of its corners are not labeled and thus not recognized as part of the region. However, such isolated pixels are not suited for our root-finding algorithm anyway, since we need at least a 2x2 region to compare two different points.
- For the contour-tracing based CCL algorithm, it is critical to know whether a given internal or external contour has already been traced. Therefore, while tracing a contour, the original algorithm marks its surrounding background pixels in the output buffer that contains the labels. This method is fine since for the original problem that labels disconnected regions in a buffer, the background pixels are not used anyway. In our case however, a pixel being a background pixel for one region, can be a region pixel for another. Marking surrounding pixels is therefore not suitable for our problem. Thus, we set our mark directly at the pixel itself. We therefore use the remaining unused bit in the curvature map byte setup to store whether a contour pixel was already processed (P-flag, see Figure 4.4).

4) Labeling suppressed contour pixels In the last step, in order to obtain fully labeled regions that are not separated anymore, we perform a simple pass on the LHC map that assigns to each suppressed pixel the label of (a)

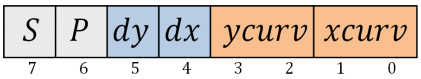


Fig. 4.4: Full byte setup of the curvature map during labeling. The S-bit marks a pixel as suppressed, the P-bit indicates contour-pixels that have already been processed.

its right neighbor or (b) its bottom neighbor, depending on the availability of labels at the neighboring pixels.

4.4.2 Bounding box and weight information

While labeling, each time we assign a region's label to a pixel, we increase a counter for the pixel amount and simultaneously adapt the bounding box information that is tracked for each of the regions. The screen-space corners of the bounding boxes are stored in an array, and later passed to the GPU in a 1D-texture that contains one bounding-box per texel by storing the its 4 coordinates in the texel's RGBA-channels. This 1D texture represents the *LHC bounding box map* output buffer.

After finishing the LHC map, we perform two additional calculation steps: First, we calculate the relative amount of pixels per region, i.e. a fraction value that can indicates the relative size of an LHC region. These values are considered do be *region weights*. Based upon these weights, we perform a binary insertion-sort algorithm on the region labels, which arranges the labels in an array beginning with the label of the biggest region (highest weight) in descending order. The array of sorted labels is also passed to the GPU as 1D-integer-texture (*sorted region map*). Based upon the order in the sorted region map, the regions weights are accumulated, as shown in the following example:

sorted region weights:	0.4, 0.3, 0.2, 0.1
accumulated region weights:	0.4, 0.7, 0.9, 1.0

These accumulated region weights are later used for the weighted point distribution. Along with the LHC map, the LHC bounding box map and the sorted region map, these accumulated weights are passed to the GPU in a further 1D texture, the *sorted accum. weights map*. In the next section we will see, how these buffers are used in order to distribute the scene points among the screen-space mirror regions.

4.5 Mirror G-Buffer shading

The shading of the Mirror G-Buffer is performed in three steps in order to maintain mirror visibility while being able to accumulate points in the G-Buffer:

1. Point splatting depth pass
2. Point splatting visibility pass
3. Averaging pass

The first two steps once pass all points in the scene through a vertex program that performs our screen-space root-finding algorithm in order to find the reflective pixels on which to splat the scene points. The first pass is depth-buffered, and stores only a depth map of the mirrors. The second pass then uses this depth map for visibility when rendering the actual scene information to the Mirror G-Buffer. The third pass finally averages the accumulated data to obtain an anti-aliased mirror projection of the environment.

4.5.1 Weighted point distribution

When passing the scene points through our vertex program in both the depth and the visibility pass, we distribute them among the given LHC regions based on their accumulated weights.

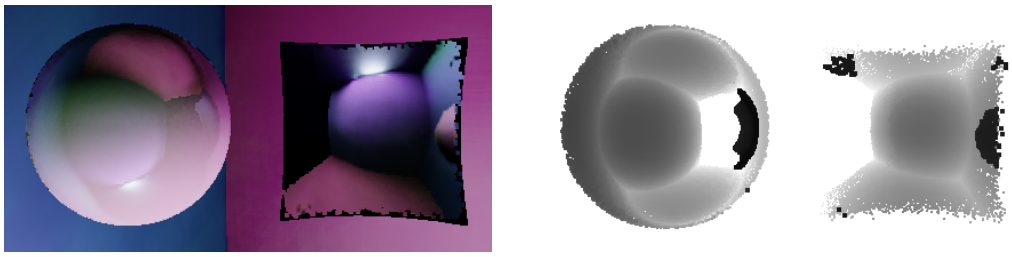
In order to perform a fast assignment calculation even for a high number of regions, we take a random value p (between 0 and 1) from a Halton sample texture and perform a binary search of p in the 1-dimensional sorted accumulated weights texture. This binary search quickly narrows down the ordered position of p among the accumulated weights. Due to the setup of the accumulated weight texture, the texture coordinates the binary search converges at can be used to look up the associated region label in the sorted region map. Performing this process on all incoming points in the vertex shader, we obtain a weighted distribution of the scene points based on the size of the LHC regions.

4.5.2 Depth pass

Input buffers:	• Camera G-Buffer [normal, SI, SP] map • Camera G-Buffer [linear depth] map • LHC map (1 channel int32) • LHC region info map (4 channel float32) • sorted LHC map (1 channel int32) • sorted LHC accum. weights map (1 channel float)
Output buffers:	• Mirror G-Buffer depth map
Parameters:	*seedAttempts, stepAttempts, maxStepIterations, mirrorZFar*

In this pass, all points are rendered to one render target by a vertex shader that uses our root-finding algorithm to splat them onto the correct mirror pixel of the Mirror G-Buffer. We therefore use the *seedAttempts*, *stepAttempts* and *maxStepIterations* parameters, which control the behavior of our root finding algorithm.

We make use of a depth buffer in order to render always only the nearest scene point that is visible in a mirror pixel. For each splatted point, we calculate its linear depth normalized to a far clipping distance, which is determined by the *mirrorZFar* parameter. This depth value is then written to the output buffer, obtaining a reflective depth map of the scene. Figure 4.5 shows such a depth reflective depth for the mirror scene in Figure 3.8.



(a) GI scene (b) Mirror depth map

Fig. 4.5: Reflective depth map for the mirror surfaces in Figure 3.8.

4.5.3 Visibility pass

Input buffers:	• Camera G-Buffer [normal, SI, SP] map • Camera G-Buffer [linear depth] map • LHC map (1 channel int32) • LHC region info map (4 channel float32) • sorted LHC map (1 channel int32) • sorted LHC accum. weights map (1 channel float) • Mirror G-Buffer depth map
Output buffers:	• Accumulated Mirror G-Buffer (3 textures) • Mirror G-Buffer counter texture
Parameters:	*seedAttempts, stepAttempts, maxStepIterations, mirrorZFar, mirrorSpaceDepthOffset*

This pass executes exactly the same point processing algorithm as the depth pass before. This time, we render to four render target textures: three textures containing the necessary Mirror G-Buffer information, and one counter texture that accumulates the number of points splatted to each pixel. In this pass, depth-buffering is disabled and blending is enabled, since we want to accumulate several scene points per mirror pixel. For each splatted point, its normalized linear mirror-space depth is calculated similar to the depth pass, but this time this depth value is used to perform a shadow comparison of a splatted point with the depth value stored in the Mirror G-Buffer depth map. This shadow comparison is performed per pixel in the fragment shader program and uses the *mirrorSpaceDepthOffset* parameter as shadow depth offset.

4.5.4 Averaging pass

Input buffers:	• Accumulated Mirror G-Buffer (3 textures) • Mirror G-Buffer counter texture
Output buffers:	• Mirror G-Buffer (3 textures)

To finish Mirror G-Buffer rendering, we perform a final pass to average the accumulated values in the Mirror G-Buffer, writing to three render targets. We draw a fullscreen quad using a fragment shader program that looks up the pixel values in each of the three accumulated Mirror G-Buffer textures, and averages them by the number of accumulated points stored in the Mirror

G-Buffer counter texture.

The result is a Mirror G-Buffer that consists of the following three 4-channel textures, containing information about the mirror surface point P and a virtual scene point Q that represents an *average* of multiple real scene points which are reflected in P:

MGB texture	Datatype	Data content
[qDiffuse | qSI]	float	**rgb:** diffuse reflection component of Q **a:** normed specular intensity of Q
[Normal | SP]	float	**rgb:** surface normal of Q **a:** normed specular power (shininess) of Q
[qPosW | pSI]	float32	**rgb:** world-space position of the reflected surface point Q **a:** normed specular intensity of the mirror surface point P

The first two textures store the surface normal and material information of the mirrored surface points in the G-Buffer. The last texture stores the world-space position of the point that is reflected (Q), and the specular intensity of the mirror point that reflects Q. The latter is used as weight when blending the mirror image component of a surface over its diffuse image component.

4.6 Pull-push closing

4.6.1 Pull pass

Input buffers:	• Mirror G-Buffer (3 textures)
Output buffers:	• 1st-4th pull level Mirror G-Buffer (4*3 textures)
Parameters:	*pullLevels*

In our implementation, we create up to 4 pull levels of the textures of the Mirror G-Buffer. The *pullLevels* parameter of this pass controls the actual pull level count (i.e. the actual height of the resulting image pyramid). Each pull level texture has half the size of the previous one, and each of its pixels contains an average value of the four pixels lying one level below in this image pyramid. However, we do not apply ordinary mip-map creation on

our textures, since we only want to average those pixels of a 2x2 pixel quad which actually contain information about mirror splatted scene points.

We thus iteratively render a fullscreen quad, each time halving the render target viewport size. In each iteration we draw to 3 render targets, one for each Mirror G-Buffer texture.

4.6.2 Push pass

Input buffers:	• LHC map • Mirror G-Buffer (3 textures) • 1st-4th pull level Mirror G-Buffer (4*3 textures)
Output buffers:	Closed Mirror G-Buffer (3 textures)
Parameters:	*pullLevels*

In this pass, pixels that are missing in the Mirror G-Buffer (i.e. containing no information at mirroring pixels) are filled with the averaged mirror information of its pixel neighbors from the higher pull level textures. Note that this procedure has the drawback that it can also assign point mappings to mirror surface pixels which actually would not reflect any scene information. In closed environments, this is problem does not occur.

Restricting the number of pull levels to 4 has a big advantage for the performance of our push operation. We do not need to perform the push pass in the same iterative way as the pull pass, i.e changing render targets and executing drawing several times. Since we deal with at most 5 Mirror G-Buffers (original G-Buffer plus at most 4 pull levels), each consisting of 3 textures, we do not need more than 16 input texture units for our shader. The 16th texture unit is occupied by the LHC map, which is used to determine whether an empty pixel in the Mirror G-Buffer actually represents a mirror pixel lacking reflection information, or just a background pixel that doesn't belong to a mirror surface. Since in our OpenGL environment we are able to use 16 input texture units per shader program, the push pass can be be performed by a single pass.

4.7 Global Illumination shading

Input buffers:	• Closed Mirror G-Buffer (3 textures)
Output buffers:	• interleaved indirect illumination accumulation buffer (3 channel float32)

After the finished and pull-push closed Mirror G-Buffer was created, it contains all information that is necessary for GI shading as described in Section 4.1: First, interleaved indirect illumination shading is applied in order to accumulate the light incident from the given VPLs in the scene. Then, this interleaved accumulation buffer is merged and filtered, finally adding direct illumination.

The sole difference between mirror GI shading and conventional GI shading of the rest of the scene is that when evaluating the light transport from a VPL over a surface point Q to the viewpoint E, then E is not the position of the camera but rather the world-space position of the mirror surface point P associated with a Mirror G-Buffer pixel at screen coordinates c. Both the positions of P and Q are gained from the Mirror G-Buffer. P is stored in its *[wpos]* texture at c, while Q is obtained by applying the inverse camera view-projection matrix to the screen-space coordinate c. [Pre10] describes the indirect illumination shading process in detail.

The final result of the SSCR pipeline is an image showing only the visible mirror reflections of the global illuminated environment. In a final pass, this mirror image is blended over our original GI illuminated image. This blending is weighted by the material properties of the mirror surfaces, i.e by their Specular Intensity (SI) value that is available in the Camera G-Buffer.

Chapter 5

Results

5.1 Implementation and platform

Our SSCR pipeline was implemented in C++ and OpenGL. It represents an extension to the GI algorithm for our *Terapoints* point cloud renderer. Each rendering pass suitable for parallelization is executed on the GPU, implemented in NVIDIA's CG shading language. The only task that is performed on the CPU is the creation of the LHC map, since this pass implements a fast connected-component labeling algorithm, that runs at cost of $O(n)$, and is not suited for parallel execution on the GPU.

In the previous chapters, we have introduced and discussed a number of parameters, which control the behavior of the SSCR pipeline at different stages, affecting both image quality and performance. In the following, we will evaluate their influence in detail. Further, we will discuss the dependency of image quality and performance on the amount, size and complexity of mirroring surfaces in the scene.

In order to provide a meaningful evaluation on the influence of the SSCR parameters and the scene complexity on the result, we will observe different mirroring objects representing different categories of surface complexity, under equal conditions in a Cornell box test setup.

All images and performance values are taken respectively observed on a platform with an Intel Xeon X5550 2.67GHz CPU with 72 GB RAM, and a GeForce GTX 285 GPU with 1 GB dedicated video RAM.

5.2 Sample renderings

Since we cannot use any previous work on mirror reflections in point clouds that could be compared to ours, and image quality in mesh scenes is not comparable to that in splat-rendered point cloud scenes, we evaluate the influence of different SSCR parameter values by comparison with sample renderings

that individually adapt all parameters in order to provide a maximum possible image quality.

We use five different representative models with a shininess attribute set to maximum, i.e. exhibiting mirroring surfaces. Our models are placed in the horizontal and vertical center of a closed stretched Cornell box, which is lit by a spotlight that is orientated towards the lower right corner of the box. The camera is placed inside the box near the front vertical wall, containing the object the center of its view frustum. The models are placed in a distance to the viewer which avoids that they are directly lit by the spotlight. This is to ensure an identically illuminated environment. Figure 5.1 illustrates this setup.

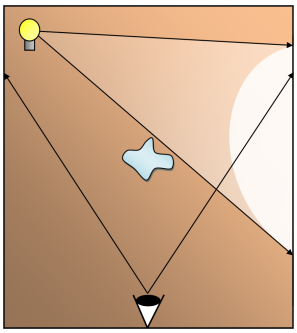
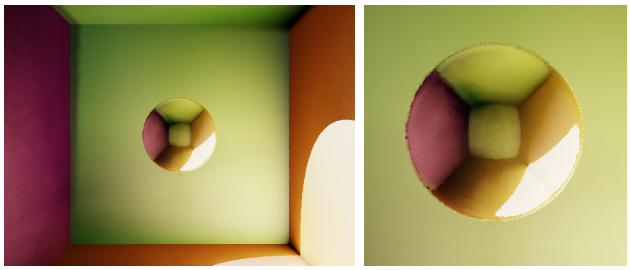


Fig. 5.1: Schematic setup for our test scenes.

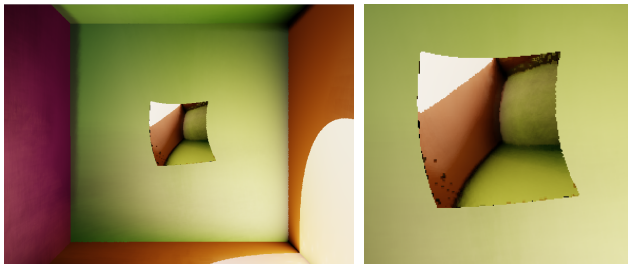
The viewport size is 800 x 600 pixels. Our models cover approximately 8-10% of the framebuffer. We render the GI scenes using 256 VPLs and one indirect light bounce. We do not apply interleaved indirect illumination shading, i.e. each pixel in the scene is illuminated by each VPL and there is no merging or filtering of the GI shaded G-Buffer. This slows down performance a bit, but on the other hand produces sharp GI shaded mirror images, allowing us to observe the work of our SSCR algorithm closely. Depending on the mirror object located in each scene, the total number of scene points that are used for Mirror G-Buffer splatting lies between 1.5 and 1.8 million points. The SSCR parameters are chosen for each model individually in order to achieve an approximated optimum image quality. All performance values are taken from snapshots of the application, i.e. they are not averaged over several frames. We use them as suitable representatives of the temporally coherent global performance achieved in our test scenes.

Figure 5.2 shows our five representative models, and lists the SSCR parameters by which they were rendered. *Max. splat size* is the maximum splat size for splatting points onto the mirror surface, *threshold accuracy* is the threshold pixel accuracy for the root-finding reflection point calculation.



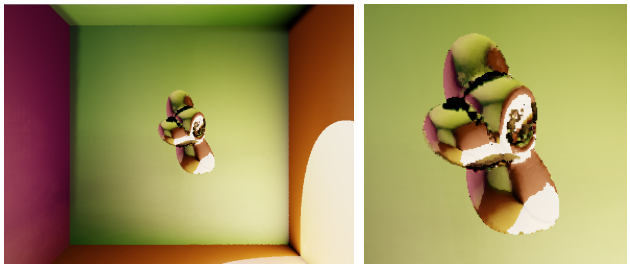
regions:	1
hit rate:	35%
max. splat size:	2
threshold accuracy:	2
pull-push levels:	4
initial step size factor:	0.2
seed/step attempts:	6/6

(a) Sphere



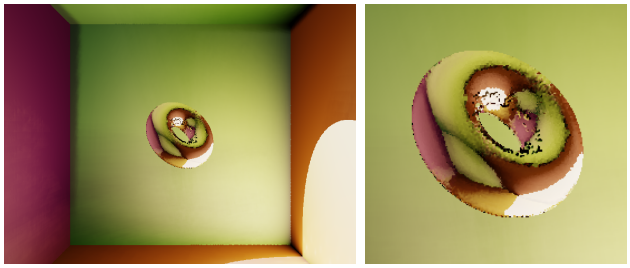
regions:	1
hit rate:	19%
max. splat size:	3
threshold accuracy:	4
pull-push levels:	4
initial step size factor:	0.2
seed/step attempts:	6/6

(b) Parabolic mirror



regions:	12
hit rate:	25%
max. splat size:	3
threshold accuracy:	4
pull-push levels:	4
initial step size factor:	0.2
seed/step attempts:	6/6

(c) Blob



regions:	2
hit rate:	17%
max. splat size:	2
threshold accuracy:	3
pull-push levels:	3
initial step size factor:	0.2
seed/step attempts:	6/6

(d) Torus

regions:	279
hit rate:	11%
max. splat size:	2
threshold accuracy:	2
pull-push levels:	4
initial step size factor:	0.2
seed/step attempts:	6/6

(e) Bunny

Fig. 5.2: Renderings of our five representative mirroring objects. (Bunny model courtesy of Stanford Computer Graphics Laboratory)

The *hit rate* shown for each scene is the percentage of scene points for which a reflection point was successfully found. Note that this rate is clamped by nature of this closed Cornell box scene since there are many points which cannot be reflected by any mirror surface. However, the uneven hit rates between the models is mainly due to their different complexity.

Sphere Convex object representing the simplest curved mirror surface containing one single homogeneously curved region. The reflections produced on its surface are similar to those which can be produced on environment mapped spheres.

Parabolic mirror Concave object that simulates a parabolic mirror surface like a shaving mirror or a teaspoon. It also consists of only one region of homogeneous curvature. Characteristic for this surface is that it flips the mirrored image if far enough away from the viewpoint.

Blob Artistically formed object containing several bumps (convex) and dents (concave).

Torus Object containing 1-2 regions (based on the viewing angle), though containing a hole. This object especially challenges the root-finding algorithm due to the chance of stepping into the hole.

Bunny Representative model for highly complex surfaces containing a high number of very small regions. The main challenge on this surface is to find reflection points despite the small pixel diameter of its regions. A too high initial step size could let the algorithm instantly step outside the region, while a too low initial step size could result in the step size being reduced to zero before success.

5.3 Overall performance

Figure 5.3 shows the overall performance of our SSCR renderer for our five scenes, measured in milliseconds of computation time. The left graph shows the computation time for the complete SSCR pipeline, starting with curvature and LHC map creation (red) and ending with the GI shading of the scene (orange). In this graph, curvature map and LHC map creation are combined to one step. A close performance evaluation of these two steps

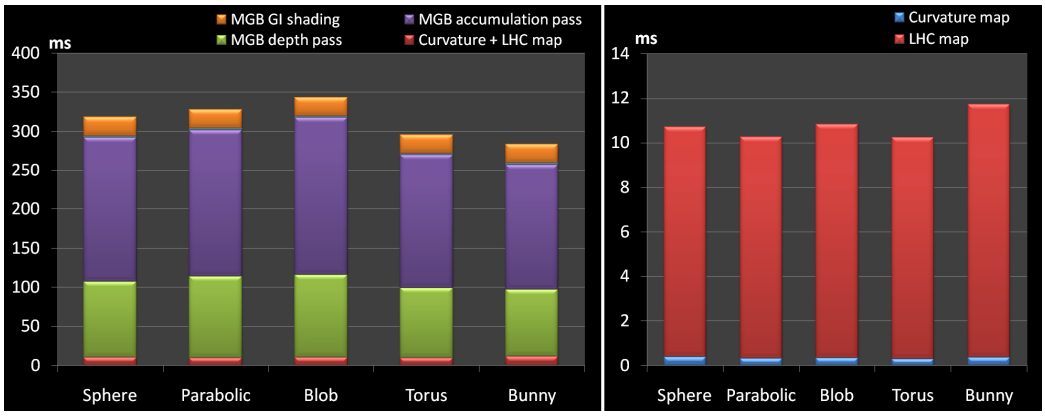


Fig. 5.3: Left: Computation time of the SSCR pipeline in milliseconds for our scenes in sequential order: curvature + LHC map (red), MGB depth pass (green), MGB accumulation pass (purple) and GI shading pass (orange). Note that before GI shading, there is also a very short pass for averaging and pull-push closing the MGB (lightblue). Right: Closeup on the computation time of the curvature map (blue) and the LHC map (red).

is given by the right graph. Note that besides labeling, the LHC map pass (red) also incorporates downloading the curvature map from the GPU RAM to the system memory, and loading back the finished LHC map to the the GPU again.

Curvature map creation (0,35ms) and GI shading (25ms) are independent of the scene content, since they are simple per-pixel operations in screen space. The LHC map computation time is evenly settled between 10 and 12 ms, however showing a slightly higher outlier for the bunny scene. This is due to the bunny's high number of regions that have to be labeled by the contour-tracing-based CCL algorithm. Many small regions require the algorithm to perform more contour tracing steps. Further, a higher amount of bounding box information is produced by the labeler which has to be loaded to the GPU afterwards.

The only rendering pass that varies between the different scenes is the Mirror G-Buffer (MGB) pass, which also clearly requires most of the computation time (250 up to 300 ms). Note that each of the 1.5M points is rendered to the Mirror G-Buffer twice, due the two-pass visibility algorithm that is used for point accumulation in the MGB. However the first pass, which just renders the points to a depth map (1 render target), needs approximately only half the time of the second pass, which renders to 4 render targets (4 MGB textures) and additionally has to perform a texture-lookup for each fragment (depth-comparison). Thus, relinquishing the smooth mir-

ror mappings obtained by point splat accumulation, and using the nearest point method for visibility, would roughly double our frame rates.

The bunny model, showing the highest surface complexity (279 regions), is rendered fastest, since due to its small regions the root finding converges more quickly, either finding a reflection pixel or loosing the point by stepping outside the region. On the other hand, the complexity of the bunny scene leads to its high point loss rate in comparison to the other objects (see Figure 5.2). The most time consuming object is proven to be the blob model. It contains only 12 regions, but still needs 60 ms longer for MGB computation than the bunny. On the other hand, its pixel hit rate is far below that of the sphere scene, which is rendered faster than the blob, although it contains the largest region to search a reflection point within. We can conclude from this that neither highest scene complexity nor highest region sizes alone are most performance drawing, but rather a mixture of both factors, like given in the blob scene, can be most challenging for the algorithm.

In the following, we will evaluate the influence of several SSCR parameters on image quality and performance based on selected surface types from this list.

5.4 SSCR parameters

5.4.1 Seed and step attempts

As discussed earlier, the *seedAttempts* and the *stepAttempts* parameters have similar functions. They both reduce the number of points lost when performing the screen-space root finding. The *seedAttempts* parameter reduces the number of points seeded outside the intended region, while the *stepAttempts* parameter avoids too much pixels from stepping out of its region.

Figure 5.4 illustrates the point hit rate and the Mirror G-Buffer computation time in the sphere, torus and bunny scenes for different seed attempts. With increasing seed attempts, the point hit rate converges towards some individual maximum for all objects. The same holds for the MGB computation time, though each additional attempt costs several milliseconds in every scene. Note that the computation time for the sphere scene increases faster than for the other scenes. This is an indicator for the overall higher number of steps in the large screen-space region produced by the sphere. This means that in the sphere region more points are successfully seeded in relation to the other scenes. This therefore requires a higher number of steps and thus computation time, but in the end not contributing progressively more hits. We can conclude from this that the larger a LHC region is, the less efficient

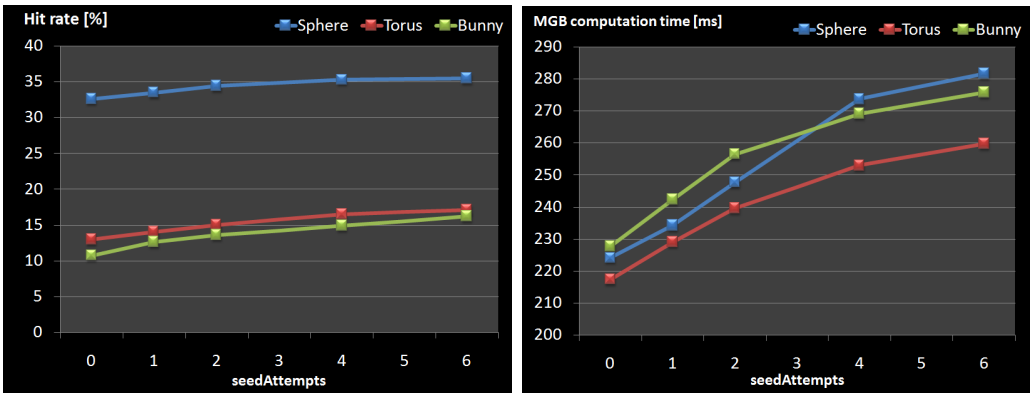


Fig. 5.4: Influence of the number of seed attempts on the hit rate (left) and the MGB computation time (right) in the sphere, torus and bunny scenes. Hit rate and performance values were measured for 0, 1, 2, 4 and 6 seed attempts, keeping all other SSCR parameters the same as for the test scenes.

are additionally applied seed attempts. Looking at the step attempts in Figure 5.5, we also encounter a convergence of the hit rate. In case of the sphere and the torus, our application even reports a reduction of successful points. However, this could be caused by a slight imprecision of the performance tracking in our application. As we see, the biggest improvement is achieved by the first additional step attempt. After the second attempt, no further benefit is drawn for simple objects (sphere and torus) while the bunny still gains additional hits. On the other hand, for the bunny the additional MGB computation time grows more intensely than for the simpler mirror surfaces. This is due to the high need for additional steps when dealing with a high number of small, complexly shaped screen-space regions.

In order to evaluate the effect of these parameters on image quality, we observe the bunny scene, which is affected the most, and reduce the pull-push closing iterations to 1. This allows for better observation of the pixel densities achieved by the additional seed and step attempts. Figure 5.6 simultaneously changes both the seed and the step attempts and shows point hit rate and the resulting appearance of the bunny. According to the graphs in Figures 5.4 and 5.5, the greatest benefit for pixel density is drawn by the first two additional attempts.

5.4.2 Initial root-finding step size

The most significant performance parameter for the root-finding algorithm is the choice of its initial pixel step size. We have already discussed that

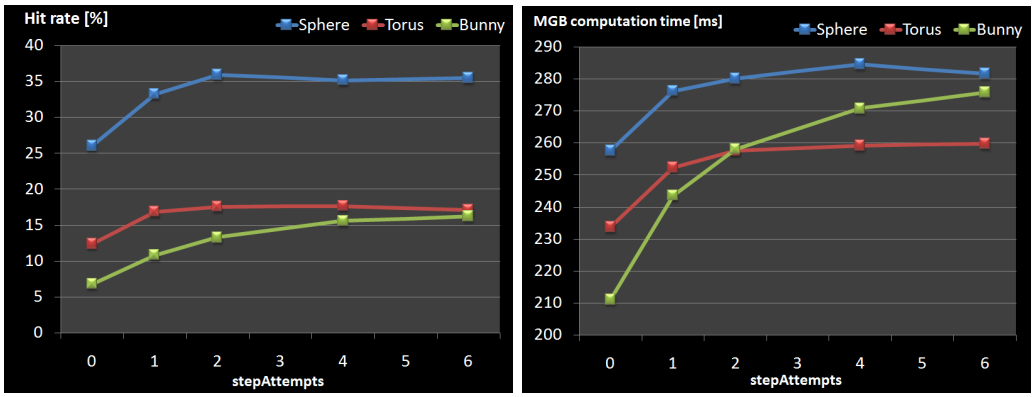
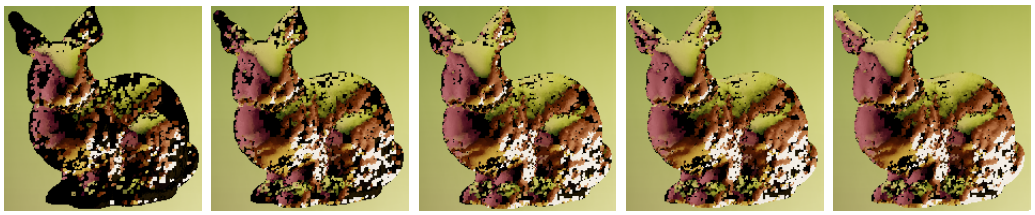


Fig. 5.5: Influence of the number of step attempts on the hit rate (left) and the MGB computation time (right) in the sphere, torus and bunny scenes. Hit rate and performance values were measured for 0, 1, 2 and 4 seed attempts, keeping all other SSCR the same as for the test scenes.



(a) 0/0: 7.39% (b) 1/1: 4.34% (c) 2/2: 7.87% (d) 3/3: 9.95% (e) 5/5: 12.16%

Fig. 5.6: Image quality of the bunny scene depending on the maximum seed and step attempts. The shown percentage is the point hit rate achieved with the respective settings.

this step size is reduced throughout the process, each time we overstep the reflection point. Generally, the bigger the initial step size is chosen, the less steps are necessary to find a reflection point in large LHC regions. A too wide step size increases the risk to lose points by stepping out of the region. On the other hand, too small step sizes let the algorithm converge much slower, which reduces performance.

We now want to evaluate the influence of the initial step size on the general number of steps the algorithm needs to converge. As representative test setup we limit the number of steps to 10, observing the percentage of points that have not converged. Note that this implies a reduction of computation time that does not tell much about the parameter's influence on the overall performance. We choose the sphere and the torus scenes for our test, since these two objects provide large screen-space regions where the

influence of the step-size on the necessary steps is recognizable. The torus represents an interesting counterpart of the sphere. Since its LHC region contains a hole, the step size is also expected to influence the number of points lost by stepping into it. Figure 5.8 shows the percentage of scene points not converging within the first 10 iterations for various initial step size factors. Note that the initial step size factor represents the initial step size as fraction of the width of the LHC region in which a point is seeded.

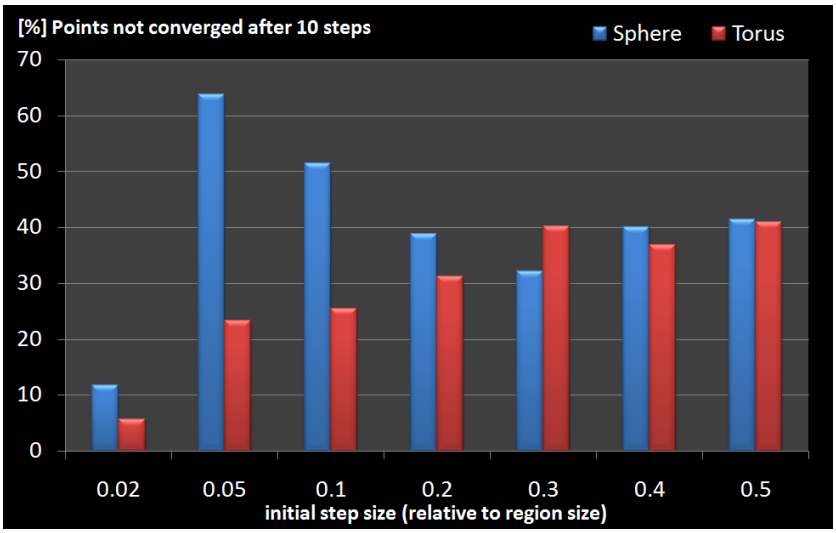


Fig. 5.7: Percentages of points requiring more than 10 steps, based on different initial step size factors.

For the sphere scene, the figure shows a complex dependence between initial step size factor and point convergence. Between 0.05 and 0.30, the number of points not converging after 10 steps decreases with increasing step size. This matches our expectation that higher initial step sizes lead to a faster convergence of the root-finding algorithm. However, between 0.30 and 0.50 the step count required for convergence increases again in the sphere scene. This shows us that the number of necessary steps can also increase if the step size is too high, because in this case the algorithm has to perform several zic-zac steps around the reflection point before the step size is sufficiently reduced down to allow convergence. On the other hand, an initial step size factor of 0.02 shows an extraordinarily fast convergence behavior, which doesn't seem to fit in either of our previous two categories. This is explained by the fact that a too low initial step size loses the scene point very quickly by being reduced to zero after the first few step size reductions.

Looking at the torus scene, we observe a completely different, but less complex dependence on the step size. Increasing the step size almost contin-

uously increases the number of steps the algorithm needs to converge. This is due to the fact that the torus – although containing only two LHC regions – shows a much more complex, higher frequent surface curvature than the sphere. Thus, for this object smaller step sizes are better suited since they approach the reflection point more carefully.

In order to evaluate the influence of the initial step size on the overall performance, we run another test series. This time we do not clamp the maximum iteration count, but rather let the algorithm step until either a reflective pixel was found or the pixel was lost. Further, we increase the screen area occupied by the mirror regions in order to magnify the effect of this factor on the performance. We use the sphere and the bunny models for our observations, moving the camera close to the objects to achieve a screen coverage of approximately 50%. Figure 5.8 shows the resulting performance graph for several initial step size factors for these scenes.

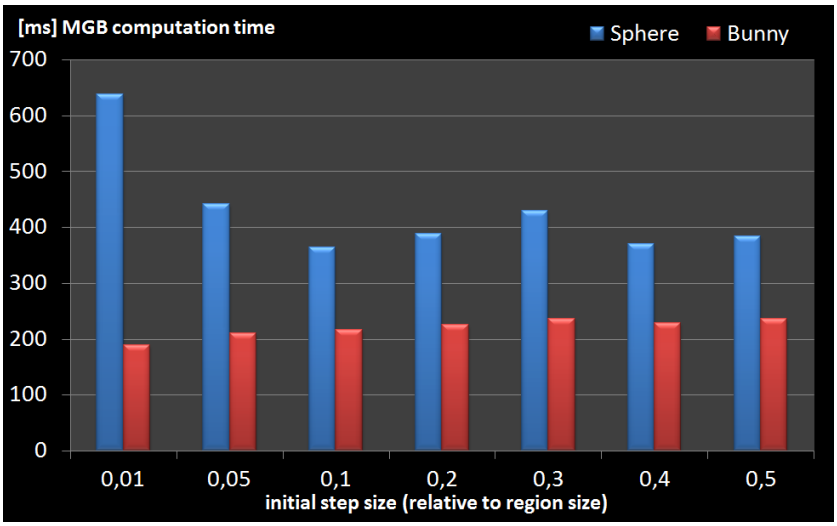


Fig. 5.8: MGB computation time for different initial step size factors in closeup scenes of the sphere model (1 LHC region) and the bunny model (775 LHC regions). Both closeups scenes contain approximately 50% of mirroring viewport pixels.

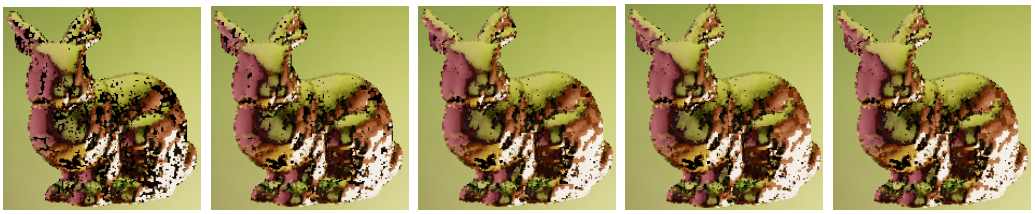
The closeup sphere scene shows a similar behavior than the normal sphere scene from the previous step size test. Up to a certain step size factor (in our case 0.1), the performance for the Mirror G-Buffer rendering is enhanced when increasing the step size. For step sizes above this critical value, the computation time evens out around some average value (400 ms). In contrast to that, performance in the closeup of the bunny scene doesn't depend much on the initial step size factor. This is easily explained by the fact that the bunny model contains a high number of small regions. Despite of the closeup,

these regions still lead to a fast convergence of the root-finding algorithm. Generally, comparing both graphs we see that a more detailed mirror surface provides even better performance than a less complex reflector.

5.4.3 Pull-push iterations

The bunny scene is also best suited for the demonstration of the influence of the used pull-push levels on the image quality. We observe it under the same parameters as in the original scene, only varying the pull-push count (see Figure 5.9).

As we clearly see, image quality respectively mirror pixel density is enhanced with every step, while the computation time for the pull-push pass stays below one millisecond. Therefore, the pull-push closure pass is one of the most powerful and important parts for quality enhancement in our SSCR algorithm.



(a) 0 iterations (b) 1 (0.72ms) (c) 2 (0.89ms) (d) 3 (0.92ms) (e) 4 (0.97ms)

Fig. 5.9: Influence of the pull-push iteration count on image quality in the bunny scene. The bracketed time values show the milliseconds used for the pull-push pass in each case.

5.5 Screen coverage and viewport size

We now will evaluate the influence of the viewport size and the screen coverage of mirror pixels on the overall performance of the SSCR pipeline. Figure 5.10 shows the performance graph. We use the bunny scene and observe it at 640 x 480, 800 x 600 and 1024 x 768 resolution. For each viewport size, we once capture the bunny at the distance given in the scene, and once at closeup distance (30% screen-space coverage).

We observe a simple dependence of computation time on both the viewport size and the screen-coverage. Bigger viewports result in more data to be loaded between GPU RAM and system RAM for LHC map creation. On same scenes, they also contain more mirror pixels to be shaded by the GI framework.

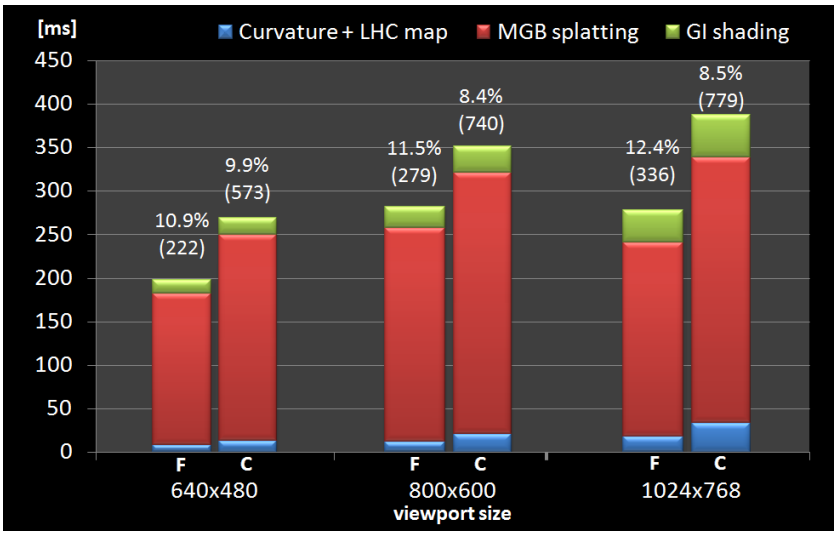
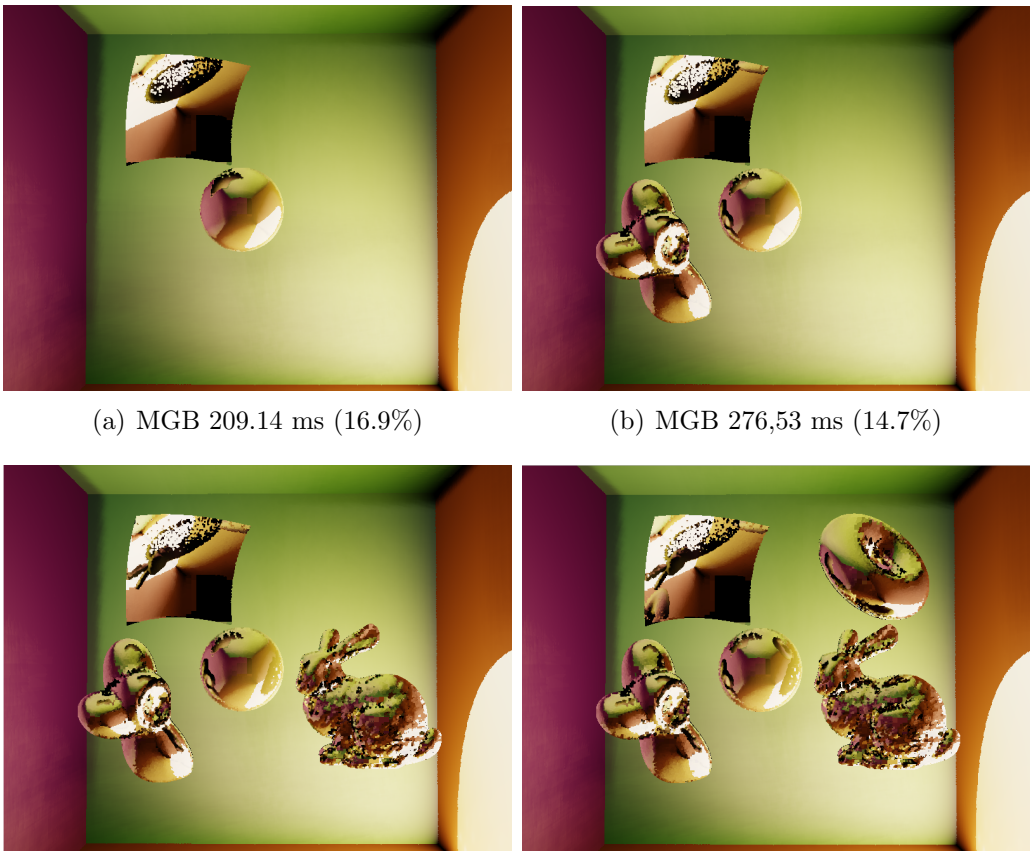


Fig. 5.10: SSCR computation time dependency of viewport size and screen-coverage. For each viewport size, the left bars shows the performance for the far bunny (F, 8% coverage), the right ones show the a closer bunny (C, 30% coverage). For each case the graph also shows the hit rate, and the number or regions in brackets.

5.6 Scene complexity

So far, we only observed single mirroring objects. We evaluated image quality and performance of various parameters based on their different characteristics. Let's now see how multiple mirrors interact with each other in a scene. In Figure 5.11, we place all our models within one scene next to each other. All scene points are used once for mirror-splatting, distributing them between each visible homogeneous mirror region. This way, we automatically achieve reflections of the mirror objects in each other. Note that so far, we only reflect the diffuse component of an object on a mirror. Since most of our models do not contain any diffuse component, their mirror image appears mostly black. This black mirror-mapping is also visible at self-reflected parts of the object's surfaces, i.e at their concave corners (see the blob or the bunny test renderings).

Following the mirror mapping of the sphere on the parabolic mirror throughout the image series, we can clearly track its pixel density reduction as we add more mirrors to the scene. The more mirror regions we add, the less scene points are assigned to an individual region. This pixel density problem could be reduced by an additional mirror splatting pass, that again distributes the scene points over the surfaces, only this time using a different



(a) MGB 209.14 ms (16.9%) (b) MGB 276,53 ms (14.7%)

(c) MGB 323,93 ms (15.4%) (d) MGB 354,87 ms (14.6%)

Fig. 5.11: Series of snapshots continuously adding mirroring objects of different geometric characteristics to the scene. The caption of each picture provides the Mirror G-Buffer computation time and its hit rate.

distribution. The second reason for the bad pixel density of the sphere image in the parabolic mirror is a low pixel splat size. Increasing the pixel splat size could condense the mirror image again. On the other hand, a higher pixel splat size would lead to a coarser, smudgier mirror mapping on more detailed objects like the bunny. In general, we see that the application of a global SSCR parameter set on a scene containing different mirror surface types is not sufficient to obtain a maximum image quality.

This points the way for the future work to be done on this renderer. To optimize the visual results of our approach, the way to go would be the implementation of a mechanism that intelligently adjusts the SSCR parameters based on screen-space mirror scene descriptors like LHC region count, bounding box sizes, etc.

Chapter 6

Conclusion

We have introduced a new technique called *screen-spaced curved reflections* (SSCR) to render mirror reflections on curved surfaces in large point clouds. We have shown a way to perform a fast 2D screen-space root finding on a *mirror-space error function*, in order to find a reflective pixel for a given scene point in the viewport.

The algorithm is not carried out completely on the GPU, since we perform a fast segmentation and a connected-component labeling algorithm for the framebuffer, which is not parallelizable. Although this requires us to copy framebuffer data between GPU RAM and system RAM each frame, this method is still very efficient for our task and does not represent a performance issue at all. We have seen that the most costly part of our method is mirror reflection rendering, which required about 350 milliseconds in our test scenes. The problem is that this part of the rendering pipeline actually is a two-pass algorithm, which does the time-consuming point splatting twice per frame, in order to perform visibility tests despite of point accumulation. In our implementation, each available point in the scene is splatted onto the mirroring surface pixels of the framebuffer. For our small Cornell box test scenes, these are already about 1.6 million points, each one stepping through the framebuffer in order to find its reflection point.

Depending on the number and type of mirror surfaces in the scene, different parameter settings of our algorithm result in a different behavior in performance and image quality. In most cases, better image quality is paid with computation time, making it hard to state globally ideal values for these parameters.

Although our algorithm does not guarantee to produce perfect mirror images, it provides a geometrically correct approach, which also handles difficult mirror surfaces like highly detailed surfaces or concave mirrors. In contrast to polygon scenes, we do not expect to gain photorealistic images from point splat scenes, rather we try to enhance the overall appearance of the scene and increase the believability of the images.

Future Work Although our algorithm is designed for point cloud scenes, we can think of possible applications to polygon scenes. We have seen that current polygon-based mirror rendering techniques, which try to exploit the fast feed-forward pipeline of current GPUs, also require finding a correct mirror mapping for each vertex of the (often pre-tessellated) scene objects before rasterization. Performing this mapping in screen-space using our SSCR approach would remove the need for any world-space data structure like sample based camera BSP trees. It would be an interesting task for the future to compare our approach introduced in polygon scenes to current techniques. Looking at current real-time global illumination (GI) approaches for polygon scenes, there are methods that depend on a point sampling of the scene object's surfaces. These point samples are used for storing the positions of virtual point lights in the scene, or for splatting on parabolic maps when creating imperfect shadow maps. In such scenes, if points are already available for GI rendering, they could also be used for SSCR computation, though their number and distribution over the scene would be an issue to evaluate.

Our method is intended to support multiple mirror bounces. We have shown how these can be achieved by a simple iterative extension of the algorithm, which yet has to be implemented. Another issue to focus on would be the introduction of an intelligent, adaptive point set selection used for mirror splatting. Many scenes containing smaller mirror surfaces of low complexity would suffice using only an interleaved subset of the points available in our scenes, i.e. passing only each 2nd, 3th or 4th point through our vertex program. Up to now, we simply pass all points in the scene through our vertex program and try to find a reflective pixel for them. Further, we expect a significant performance gain by applying simple culling techniques to our scenes in order to reduce the number of unnecessarily processed points in our

algorithm.

Another interesting work for the future would be the evaluation of other reflection point finding algorithms. We have shown that we can formulate the deviation between a reflection vector and the eye-vector as a continuous function δ over the surface, which contains the reflection points in its roots. We perform a fast 2D screen-space root-finding algorithm, which approximates the 2D problem by approaching it by two individual 1D bracketing and bisection algorithms in each screen-space direction. However, the problem can also be formulated as minimum search. There are several optimization algorithms, which could be tested in order to find a minimum in this function, like conjugate gradient or the simplex method. It would be interesting to evaluate, whether additionally necessary lookups could be outweighed by a faster convergence to the reflection point.

Enhancing the power and efficiency of our new method by following these ideas could be an important way to go in order to reveal even better, more powerful rendering methods based upon our idea in the future.

List of Figures

Bibliography

[BApS02] Stefan Brabec, Thomas Annen, and Hans peter Seidel. Shadow mapping for hemispherical and omnidirectional light sources. In *In Proc. of Computer Graphics International*, pages 397–408, 2002.

[Bli77] James F. Blinn. Models of light reflection for computer synthesized pictures. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 192–198, New York, NY, USA, 1977. ACM.

[BN76] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Commun. ACM*, 19(10):542–547, 1976.

[CCL04] Fu Chang, Chun-Jen Chen, and Chi-Jen Lu. A linear-time component-labeling algorithm using contour tracing technique. *Comput. Vis. Image Underst.*, 93(2):206–220, 2004.

[CT81] Robert L. Cook and Kenneth E. Torrance. A reflectance model for computer graphics. In *SIGGRAPH '81: Proceedings of the 8th annual conference on Computer graphics and interactive techniques*, pages 307–316, New York, NY, USA, 1981. ACM.

[DB97] Paul J. Diefenbach and Norman I. Badler. Multi-pass pipeline rendering: realism for dynamic environments. In *I3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 59–ff., New York, NY, USA, 1997. ACM.

[DBB02] Philip Dutre, Kavita Bala, and Philippe Bekaert. *Advanced Global Illumination*. A. K. Peters, Ltd., Natick, MA, USA, 2002.

[Gre86] Ned Greene. Environment mapping and other applications of world projections. *IEEE Comput. Graph. Appl.*, 6(11):21–29, 1986.

[HKWB09] Miloš Hašan, Jaroslav Křivánek, Bruce Walter, and Kavita Bala. Virtual spherical lights for many-light rendering of glossy scenes. In *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, pages 1–6, New York, NY, USA, 2009. ACM.

[HS98] Wolfgang Heidrich and Hans-Peter Seidel. View-independent environment maps. In *HWWS '98: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 39–ff., New York, NY, USA, 1998. ACM.

[Kaj86] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, 1986.

[Kel97] Alexander Keller. Instant radiosity. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 49–56, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[KH01] Alexander Keller and Wolfgang Heidrich. Interleaved sampling. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 269–276, London, UK, 2001. Springer-Verlag.

[MKC08] Ricardo Marroquim, Martin Kraus, and Paulo Roma Cavalcanti. Special section: Point-based graphics: Efficient image reconstruction for point-based and line-based rendering. *Comput. Graph.*, 32(2):189–203, 2008.

[Nic65] Fred E. Nicodemus. Directional reflectance and emissivity of an opaque surface. *Appl. Opt.*, 4(7):767–773, 1965.

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

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, 1975.

[Pre10] Reinhold Preiner. Real-time global illumination in point clouds. In *CESCG 2010: Proceedings of the Central European Seminar on Computer Graphics*, 2010.

[PSM06] Voicu Popescu, Elisha Sacks, and Chunhui Mei. Sample-based cameras for feed forward reflection rendering. *IEEE Transactions on Visualization and Computer Graphics*, 12(6):1590–1600, 2006.

[RGK+08] T. Ritschel, T. Grosch, M. H. Kim, H.-P. Seidel, C. Dachsbacher, and J. Kautz. Imperfect shadow maps for efficient computation of indirect illumination. *ACM Trans. Graph.*, 27(5):1–8, 2008.

[RPZ02] Liu Ren, Hanspeter Pfister, and Matthias Zwicker. Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. In *Computer Graphics Forum (Eurographics 2002)*, volume 21, pages 461–470, September 2002.

[SDG+98] Grossman Hon Sc, William J. Dally, J. P. Grossman, J. P. Grossman, Thanks To Bill Dally, Seth Teller, Pat Hanrahan, John Owens, and Scott Rixner. Point sample rendering. In *In Rendering Techniques '98*, pages 181–192. Springer, 1998.

[SK00] László Szirmay-Kalos. Monte-carlo methods in global illumination, 2000.

[Vea98] Eric Veach. *Robust monte carlo methods for light transport simulation*. PhD thesis, Stanford, CA, USA, 1998. Adviser-Guibas, Leonidas J.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12(3):270–274, 1978.