Friedrich Schiller Universität Jena

Faculty of Mathematics and Computer Science

Visualization and Explorative Data Analysis Group

Poisson Reconstruction for Global Illumination on Surfels

M A S T E R T H E S I S

A thesis submitted for the degree of
Master of Science (M. Sc.)
in the degree program
Computational and Data Science

submitted by Antonio Noack
born 1st July 1997 in Jena, Germany

Jena, 25th November, 2022

Examiner:	Kai Lawoon
Second Examiner:	Simon Janez Lieb

1 Abstract

We propose and analyze a surface element ("surfel") based global illumination technique. Surfels are disk shaped objects, that can store information in 3D space. The surfels in this thesis store global illumination data. With our proposed method, surfels are distributed in the scene first. We try two different methods to draw the surfels, and find a method that can draw upto two million dynamic surfels at fluent frame rates in real-time in the Unity game engine. The data of the surfels is managed entirely by the GPU to avoid bottlenecks.

Then the surfels' global illumination and their finite differentials are calculated. Finally, Poisson image reconstruction is applied to reduce noise of the data. We define a weighting scheme for colors and gradients on surfels and measure its performance. Surfels are an effective way to reduce noise of path traced global illumination, but our proposed weighting scheme is not good enough for Poisson reconstruction. We suggest a better scheme for future work.

We show the differences of a range of rays-per-pixel limits to decide how many are needed. We test surfel counts from 2^{18} to 2^{22}, surfel densities and compare the technique with and without using Poisson reconstruction. As a baseline, our results are compared to a screen-space, offline, path tracing renderer.

We test a preparation step for the Poisson iteration. This introduces slight artifacts first, but accelerates the convergence on a Gaussian blurred image. When applying this step to path tracing, it is unhelpful, and even reduces convergence speed. We implement our technique as an open source package for the Unity game engine[1]. The project can be downloaded from `https://github.com/AntonioNoack/Unity-SurfelGI`.

[1] https://unity.com/

2 Zusammenfassung

Wir stellen eine neue Methode zur Berechnung von globaler Beleuchtung basierend auf Oberflächenelementen ("Surfel") vor. Surfels sind scheibenförmige Objekte, die Informationen im 3D-Raum speichern können. In dieser Masterarbeit speichern die Surfels Beleuchtungsinformationen. Im Gegensatz zu existierenden Methoden, speichern unsere Surfels nur die Beleuchtungsinformation für einen kleinen Bereich von Materialien, und für eine feste Oberflächenausrichtung. Unsere Methode verteilt zuerst die Surfels in der Szene. Zwei Methoden werden zum Zeichnen der Surfels ausprobiert. Eine der beiden kann bis zu zwei Millionen Surfel bei flüssiger Bildwiederholrate zeichnen. Vor jedem Neuzeichnen der Surfels wird die gleichmäßige Verteilung aufrechterhalten. Um einen Flaschenhals zwischen der CPU- und der GPU-Seite zu vermeiden, werden die Surfels ausschließlich von der Grafikkarte verwaltet.

Nachdem die Surfels gezeichnet wurden, werden die Beleuchung in der Nähe der Surfels und deren finite Differenzen berechnet. Schließlich wird Poisson-Bildreconstruktion verwendet, und Rauschen im Ergebnis zu reduzieren. Wir definieren eine Interpolation der Surfelbasierten Beleuchung und ihrer Ableitung, und messen ihre Konvergenz. Surfels sind ein effektiver Weg, um das Rauschen vom Path-Tracing zu reduzieren, aber unsere Interpolationsmethode der finiten Differenzen ist nicht gut genug, um Poisson-Rekonstruction nutzen zu können. Wir schlagen daher ein besseres Verfahren für zukünftige Arbeiten vor.

Wir zeigen außerdem die Unterschiede zwischen den Ergebnissen bei verschiedenen Strahl-Berechnungslimits, um abschätzen zu können, was ein gutes Limit ist. Surfelmengen von 2^{18} bis 2^{22} und verschiedene Surfel-Dichten werden verglichen. Die vorgestellte Methode wird dabei mit den Ergebnissen der Poisson-Rekonstruktion, und einer pro-Pixel-berechneten, konvergierten Kontrollmessung verglichen.

Neben unserer Arbeit an Surfels stellen wir auch eine Idee zur Beschleunigung der Poisson-Rekonstruction vor. Diese Methode führt sichtbare Abweichungen im Bild herbei, aber beschleunigt dann die Konvergenz auf unserem Testbild. Um ungenaue Ergebnisse zu simulieren, wurde das Testbild dafür zuerst mit einem Gaußschen Kernel verwischt. Wenn man diesen Schritt jedoch auf den Ergebnissen des Path-Tracings anwendet, behindert er

die Konvergenz. Wir implementieren all unsere Methoden als quelloffenes Paket für die Unity Spiele-Engine[1]. Das Projekt kann von `https://github.com/AntonioNoack/Unity-SurfelGI` heruntergeladen werden.

[1] https://unity.com/

Contents

3 Introduction

In this thesis, we investigate combining two techniques to achieve real-time global illumination. We target real-time global illumination, so photorealistic images can be generated and interact with user-inputs, e.g., for applications like games.

Global illumination is a method to compute photorealistic images. Global illumination separates the computation of color and illumination. This allows color textures to retain sharp details, even if the illumination data is blurred. The data might be blurred, because there is limited time to calculate the solution, often a few frames.

One of those techniques that we use is called global illumination on surfels. The other one is called gradient-domain path tracing. Our work is inspired by the proprietary solution of Brinck et al. [6] and Barré-Brisebois et al. [2]. Our open source solution should inspire more future research. In the future, any game developer should be able to use surfels for real-time global illumination.

Besides that, the surfels by Brinck et al. [6] and Barré-Brisebois et al. [2] are pretty complex. Instead of using spherical harmonics to save global illumination like other techniques, we specialize surfels to their respective BSDFs (bidirectional scattering distribution functions) and make their data angle dependent. Our easier surfels simplify the implementation and shrink the storage size per surfel. As transparency is an important part of civilized environments, e.g., glass, we describe two methods to implement transparency on these surfels. We did not find a description of either of these methods for surfels in the existing literature.

The convergence behavior of the surfels and their Poisson reconstruction is measured. It is then compared to the per-pixel path traced baseline. The results show that our surfels themselves work, but their interpolated gradients aren't useful for Poisson reconstruction. We implement our experiments as an Open Source project for the Unity game engine, so it can be extended in the future.

4 Background

4.1 Raytracing vs Rasterization

In the last decades, games are typically rendered using triangle meshes in a process called rasterization. GPUs (Graphics Processing Units) have been optimized for this computation since the first as such marketed GPU, the GeForce 256[1].

Rasterization can be summarized as triangles getting projected onto the 2D image plane. Then colors are assigned to each pixel of these triangles. Rasterization is a model, that can be quite flexible and powerful if it is combined with programmable shaders.

However, in the typical rasterization model, there is only information about local scene information. Without global information, e.g., via ray-scene-traversal, photo-realistic graphics are hard to achieve.

Raytracing hardware acceleration has only been recently added to modern GPUs. Such hardware offers specialized instructions to accelerate ray-AABB and ray-triangle collision checks.

4.1.1 Rasterization pipeline

The following section describes rendering pipelines like OpenGL's and DirectX 11's pipeline. An overview of DirectX 11's and OpenGL's pipelines is presented in Figure 4.1.

The standard rendering pipeline for rasterized graphics first loads or generates polygons, lines, or points. These are stored in memory, or procedurally generated. Then they are transformed from object space into pixel space. This transformation is executed in a stage called "vertex shader" because in APIs like OpenGL, each vertex is processed separately without any means to communicate with neighbor vertices.

[1] https://en.wikipedia.org/wiki/GeForce_256

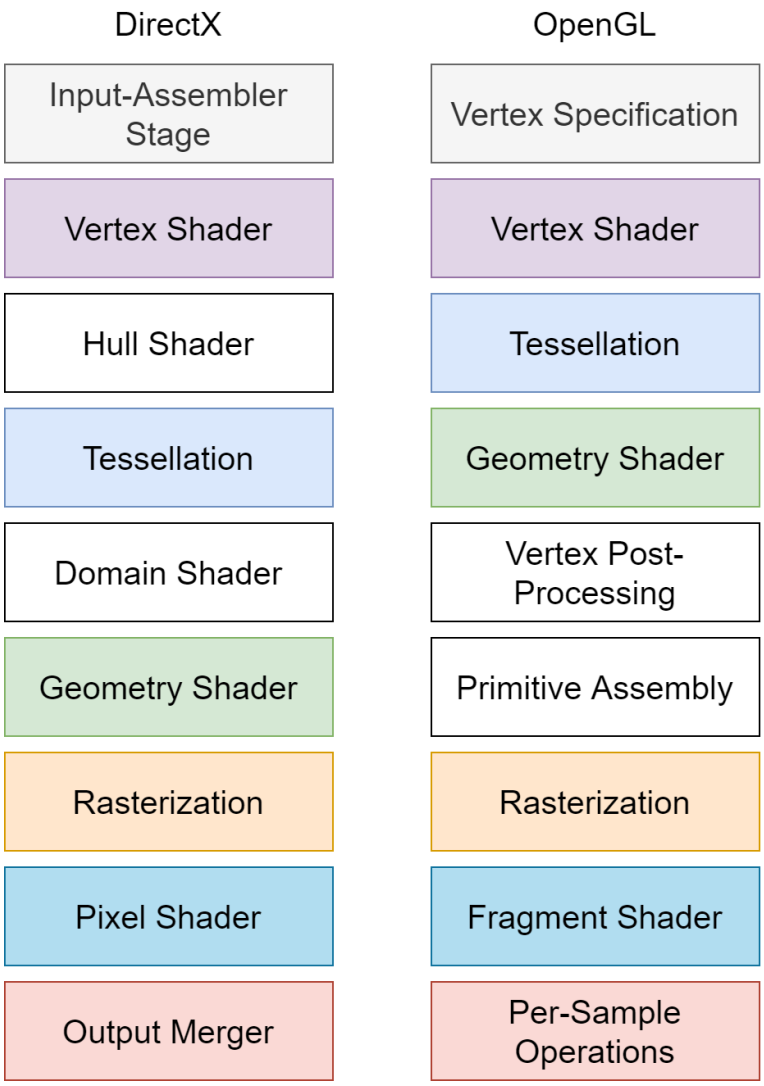

Figure 4.1 | **DirectX 11 and OpenGL pipeline.** Comparison of terminology and stages in Direct X and OpenGL. Figure was recreated in the same style, and with the same colors for equivalent stages. The original sources are `https://www.3dgep.com/wp-content/uploads/2014/03/DirectX-11-Rendering-Pipeline.png` and `https://www.khronos.org/opengl/wiki_opengl/images/RenderingPipeline.png`.

Modern APIs additionally offer triangle subdivision and mapping shaders called tessellation and geometry shader. Since triangles could overlap over the edges of the screen, the so-called rasterizer turns triangles into pixels. When a visible pixel inside a triangle has been found, there may be depth testing. Then the fragment shader (or pixel shader) is called to

determine the color at that location. Since the vertex shader may compute different shading properties for each vertex of a primitive, if these parameters are passed to the fragment shader, these parameters optionally can be linearly interpolated.

In the early days of dedicated GPUs, there was a fixed function pipeline with predefined lighting calculations. Today, shaders are fully programmable.

Optionally, each pixel can be discarded, or the depth channel can be modified. After the color has been calculated, it is blended with the destination framebuffer. Blending functions are implemented in hardware, and therefore limited. They include linear blending of colors, minimum and maximum. Typically, the blend factor is constant or defined by the alpha channel. It may originate from the calculated color (source) or destination framebuffer.

4.1.2 Compute Shaders

In pixel shaders, random access for writing values is difficult. To implement it, points instead of triangles should be drawn, and the calculation must be split between vertex and fragment shader. For each write access, the GPU must handle a point internally.

Compute shaders allow random access write operations, but lose the functionality of the rasterizer. A few new features of compute shaders include global atomic operations, local shared memory, and local synchronization. Local in this context means that it is valid within a compute group. Global means for the whole GPU.

4.1.3 Raytracing pipeline

The raytracing pipeline is implemented as a compute shader. It offers the same flexibility and features, and additionally raytracing functions.

These functions can be invoked from the shader by any subroutine of the main program and arbitrarily often. They accept a ray description, a customizable payload, and a few additional flags. The ray description defines the ray's start position, the ray direction, and the limits for the ray distance. If the upper limit is defined as infinity, the ray is therefore a line segment, and if the lower limit is negative infinity as well, it is a line.

To speed up the computation time of a ray traversal through the scene, ray acceleration structures like Kd trees are used. They have to be built before the ray queries start. The payload is used to transfer information from the hit geometry back to the compute shader.

To define that information, a pipeline stage called “closesthit” defines programmable shaders for all hit geometries. There are additional pipeline stages like “anyhit” to implement cutout-transparency, or procedural geometry: this stage decides whether the geometry inside the acceleration structure has been hit. Some stages of the raytracing pipeline allow raytracing requests as well. Recursive constructs consume stack space, so the maximum recursion depth is usually limited. Another way to speed up ray traversals is hardware acceleration. It adds new instructions, which then for example accelerate AABB-ray intersection calculations. Hardware accelerated ray tracing is only available on desktop and laptop platforms for now, and is limited to recent GPUs like Nvidia's RTX and AMD's RX 6000 series.

4.1.4 Raytracing-Rasterization-Interoperation

For calculating global illumination in pixel space using raytracing, and computing the GBuffer using rasterization, all pixel positions must be exactly inside their geometry triangle. If there are alignment mismatches, pixels are illuminated incorrectly.

In traditional path tracing (PT), the ray coordinates can be chosen randomly within the pixel, which results in simple, high-quality anti-aliasing. This matching requirement has the consequence, that other methods have to be used for anti-aliasing.

4.1.5 Ray Marching

Ray marching is a rendering method, where the scene is described using a signed distance field (isosurfaces) instead of meshes built from triangles. The scene can be traversed by calculating the distance, and iteratively advancing the current position by that distance along the ray direction. The start point is the ray origin. This process is continued iteratively until the signed distance is considered to be close enough to the surface. Such a process is illustrated in Figure 4.2. More dedicated methods like Galin et al. [9] use metrics, which take fewer iterations steps for faster convergence.

Ray marching works in any programmable rendering pipeline with enough metadata, so it can be supported on devices without hardware raytracing acceleration. A downside to ray marching is that rendering the GBuffer involves scene traversal for every pixel, which can be expensive. Another problem is convergence on surface boundaries, if a non-directional distance function is used. The distance to the geometry can be low for a lot of iterations until it is clear whether the object was hit. This problem has been illustrated in Figure 4.3.

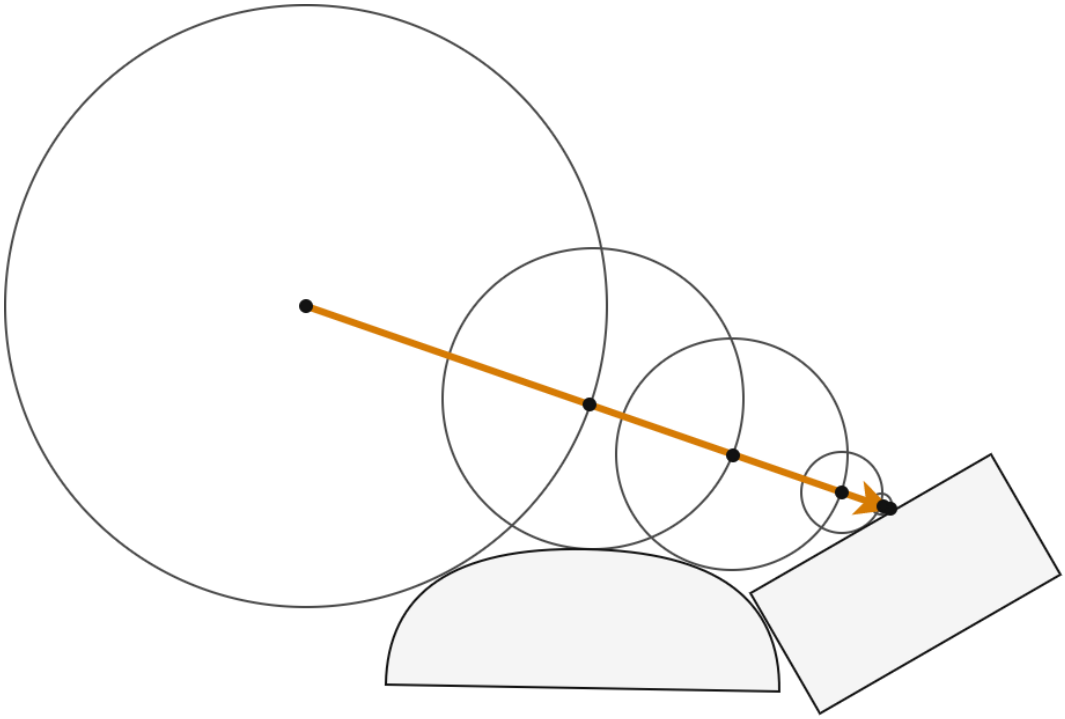

Figure 4.2 | **Illustration of the steps in ray marching.** The orange line represents the ray, and each black dot is a point, where the signed distance function was evaluated to find the step size.

There isn't a simple, well-performant method to calculate the signed distance to complex triangle meshes. Most 3D data is in triangle mesh form, so it is more difficult to find matching data for a given task. An advantage of ray marching is that mesh blending operations can be executed in real-time, like the merging from a sphere to a cube, and operations like mirroring and arrays, are cheap to integrate.

4.2 GBuffer and Deferred Rendering

There are multiple ways to render with a rasterizer traditionally. Two of them are forward rendering, and deferred rendering. In forward rendering, each pixel considers all lights within the scene for lighting. This is simple, but doesn't scale well for thousands of tiny lights, because each pixel would have to evaluate every single light.

In deferred rendering, only pixels within proximity of the light are evaluated. Instead of rendering the geometry onto a single color texture, all light-relevant attributes are rendered

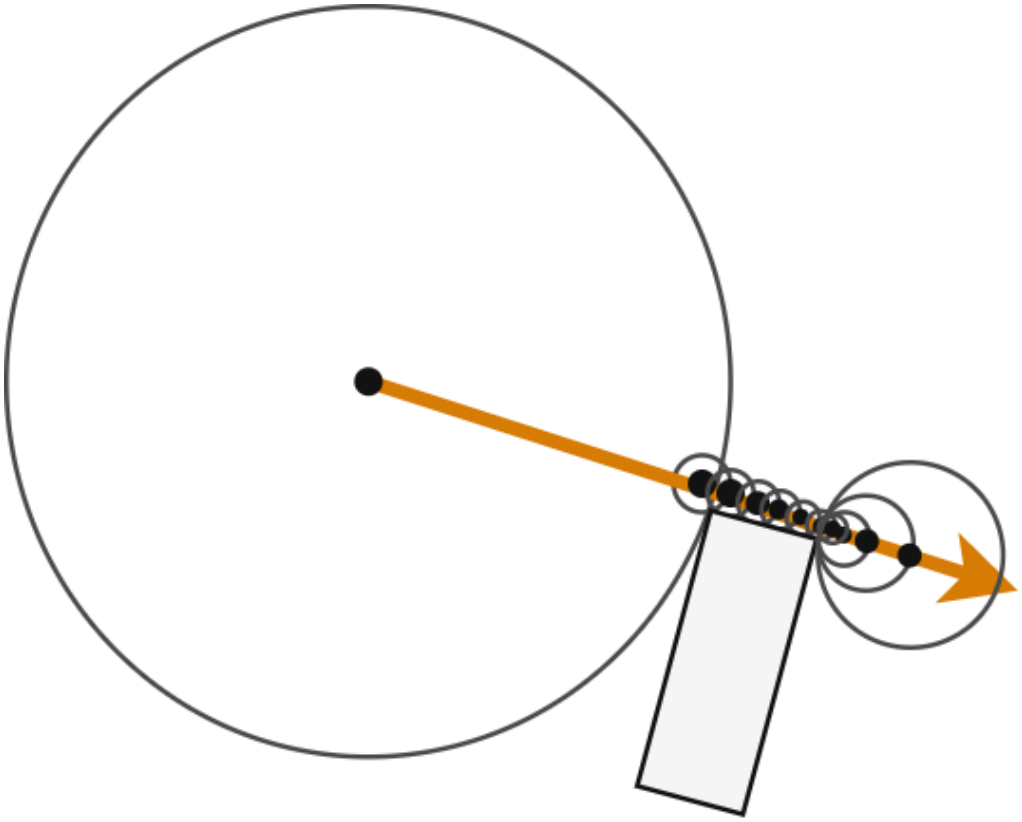

Figure 4.3 | **Illustration of the steps in ray marching close to an edge.**

onto textures. These textures are called GBuffer, and typically include albedo, normal, position, shininess, and whether a material is metallic. In the lighting stage, all lights are drawn onto a separate "light" buffer. Their region is typically defined by a box, where all fragments represent pixels that could be illuminated. Their illumination intensity then is computed based on the material attributes and the world position in the fragment shader.

4.3 Global Illumination

A commonly used model for light transport is that rays get emitted by light sources, absorbed, reflected, and refracted by surfaces until they finally reach the eye of the observer (camera). To compute light in this model, as it reaches the camera, the sum of light particles (photons) that reaches each pixel has to be determined. It can also be modeled as continuous light distributions, and it can be described by an integral. Such an integral is presented in Equation (4.1). $L(p, d)$ is the light depending on position p and direction d.

Position and direction are defined by the camera model, e.g., an image projected onto a plane. $h(p, d)$ is the closest hit from position p and direction d. A hit is an intersection of the ray with geometry at a distance greater than zero. Ω is the hemisphere of the hit surface (or black at infinity), aligned with the surface normal. *BSDF* is the BSDF in direction α, so a probability density function. The integral over a BSDF for a non-emissive material should be less than or equal to one to preserve energy. To describe colors, this BSDF can depend on the wavelength of the traced light. The term L_e has been added for emissive materials: it adds light to the scene.

$$L(p, d) = \int_{\Omega} L(h(p, d), \alpha) \cdot \mathit{BSDF}(h(p, d), \alpha) + L_e(h(p, d), d)\, d\alpha \tag{4.1}$$

In both cases, this is a difficult problem with generalized surface behaviors and scene geometry, because of its complexity. The recursive term $L(h(p, d))$ allows for complex behaviors like parallel mirrors, which produce infinite reflections. In the general case, there is no analytical solution.

In this thesis, global illumination is not described as the light, that is received by the eye, but instead the light divided by the surface (albedo) color. This color is a multiplicative, wavelength-dependent factor in the BSDF of most materials. If no such factor can be found, the surface color is defined as white (one) or an average factor for the hemisphere. This global illumination depends on the viewer's angle relative to the surface, so when the angle changes, the global illumination has to be recomputed. On diffuse, plain surfaces, the surface color can vary a lot, while the global illumination might have few high-frequency spots. When global illumination rendered to an image is blurred, because there is not enough computational power available for convergence, high frequencies in the albedo image remain intact.

4.3.1 Bidirectional Scattering Distribution Functions

Materials can be described using BSDFs (bidirectional scattering distribution functions). BSDFs are functions that describe how incoming rays interacts with a surface. Typically, a part of the light is absorbed (non-white materials), and the rest of the light continues to as light rays or particles in the world. These BSDFs describe how much light is absorbed by the surface, and how it is distributed back into the scene over the hemisphere over the surface.

BSDFs assume that the input position of a ray is the same as the output position. A more general model is the Bidirectional Surface Scattering Reflectance Distribution Function[2], where this is not enforced. For simplicity, we work with BSDFs.

There are two distinct cases of BSDFs that make it hard to find a joined model for them: A perfect mirror has a single outgoing ray direction for each incoming ray direction. The probability of each outgoing direction is zero except for the reflected direction, where it is 100% minus absorption. The probability density is zero in all directions, except for the reflected direction, where it is infinite.

Working with infinities is problematic, so renderers like Mitsuba use a merged model: The distribution of each material is split into components. A component may either be discrete, like a specific reflection direction, or continuous, like in diffuse materials.

Furthermore, discrete components can be specialized to a single ray direction. Continuous components can be separated by clusters of density, which are called "lobes". A collection of BSDF types of the Mitsuba Renderer is presented in Figure 4.4.

A diffuse material has a probability of zero for all outgoing ray directions. The probability density is non-zero.

4.3.2 Microfacet Models

Microfacet models describe surfaces of materials on the micrometer scale. Micro surfaces can be imagined as a landscape of tilted tiny plates, so called "facets". Their orientation and height determine the scattering of light on that surface. Their normal may be different from the normal of the macro-scale geometry. Facets can shadow other facets, and there can be interreflections. One such model was described by Oren and Nayar [19]. They model the surface as a collection of small, mirror-like, V-shaped cavities, as it was proposed by Torrance and Sparrow [27].

The fine ridge on the material surface are what makes materials smooth or rough, e.g., the difference between normal window glass and frosted glass, and can give anisotropic effects, as they can be found on fabrics, and the bottom of cooking pots Microfacet models can be seen as an optimization for ray traversal and memory usage, as the geometry itself could be modeled on a micrometer scale. However, in large scenes, not using a microfacet model would either limit it to use smooth materials only, or use huge amounts of geometry data.

[2] https://en.wikipedia.org/wiki/Bidirectional_reflectance_distribution_function

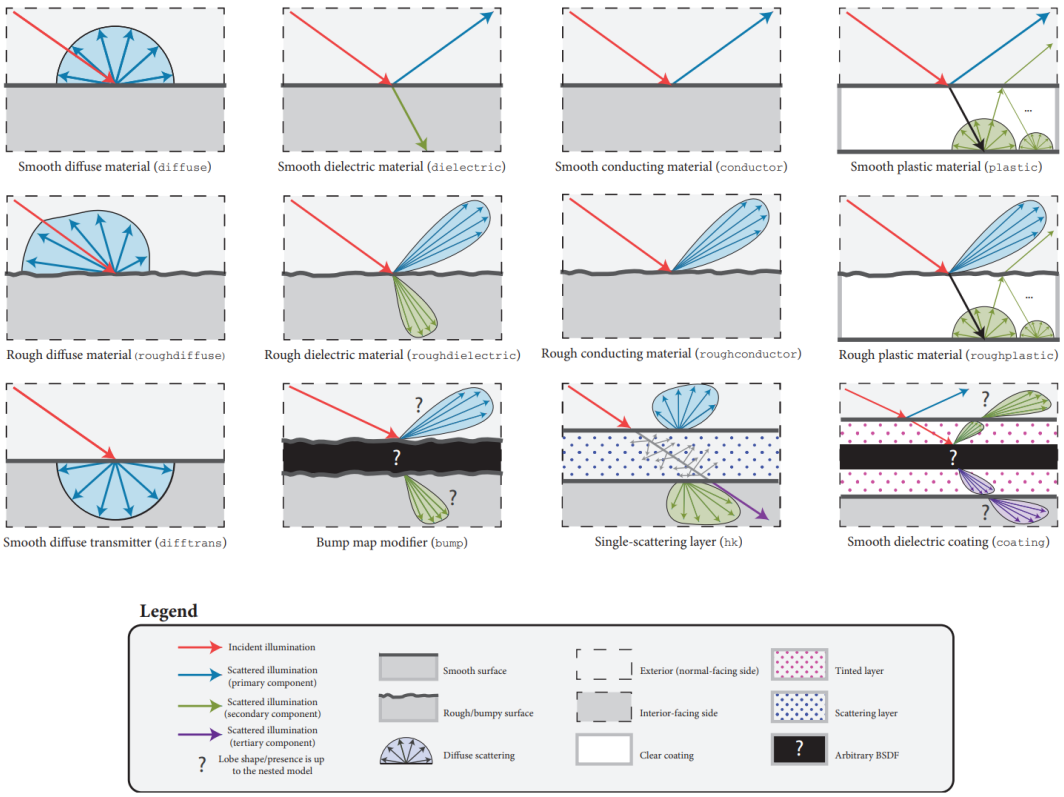

Figure 4.4 | **BSDFs.** Different material types and their schematic BSDF. This figure is taken from the Mitsuba documentation [14].

Microfacet models typically include a shadowing component, as small ridges get partially shadowed, when the light is standing at a very shallow angle.

4.4 Motion Vectors

Motion Vectors describe the previous location for each pixel in the camera view. They can be used to track the motion of geometry relative to the camera, which is useful for some techniques. One example is frame interpolation, which tries to generate frames in-between existing frames. Motion Vectors are the core component of DLSS 3 from Nvidia [18], in FSR 2.0 from AMD [23], and in temporal anti-aliasing [17]. When motion vectors are estimated from video material, they are called "Optical Flow" [29].

If global illumination is calculated per-pixel and temporal stability is assumed, motion vectors can be used to keep lighting information from previous frames, even though the camera

is slightly moving or rotating. When the camera is moving, the angle toward the surface changes, so this only works well, if the material is angle independent, or the angular movement of the camera to the surface is small. This reconstruction introduces ghosting if edges and sudden steps in depth aren't handled carefully, or the computed global illumination is angle dependent. A basic version of keeping global illumination using motion vectors has been implemented in our Unity project to compare it with our surfel approach interactively. For path tracing in pixel space, there are spatially filtering approaches like the method by Bauszat et al. [3].

4.5 Surfels

Surfels, surface elements, are a model where the surface is described by surface pieces. They are ellipses or ellipsoids in the scene, carrying data. A sample scene with sparse surfels is shown in Figure 4.5. Their area of effect or validity can be described using a radius (or their half axes) and a rotation from a standard orientation. Surfels have been used to describe 3D geometry, e.g., for point cloud rendering as a means for 3d reconstruction from video or 3d laser scans, like in Keller et al. [15] and Schops et al. [26]. For rendering, they pose the issue of hole filling: surfels are two-dimensional, circular objects, and there should not be artificial holes in the object. This introduces issues on curved surfaces (see Botsch and Kobbelt [5] and Pfister et al. [21]), as they cannot be accurately represented using flat objects. Surfels even have been used for animated models by Weyrich et al. [30] by deforming the surfels. Meshes use levels of detail. These are less detailed geometry for distanced meshes for real-time rendering. That is not that easy to implement with point clouds. Holst and Schumann [12] build surfel billboard hierarchies to represent their objects to achieve a comparable effect. L. Sang [25] uses surfel to render 3D objects on smartphones. Surfels have been used for global illumination as in Brinck et al. [6]. There, they transport global illumination data, which then is projected onto nearby geometry.

4.5.1 Elliptical Averaging

To calculate the global illumination at a pixel, data potentially needs to be interpolated from multiple surfels. If the surfels would just be drawn as circles in screen space and the surfel is tilted away from the camera, there would be very uneven overlap. Elliptical averaging is described to resolve the issues of projected surfels by Botsch and Kobbelt [5].

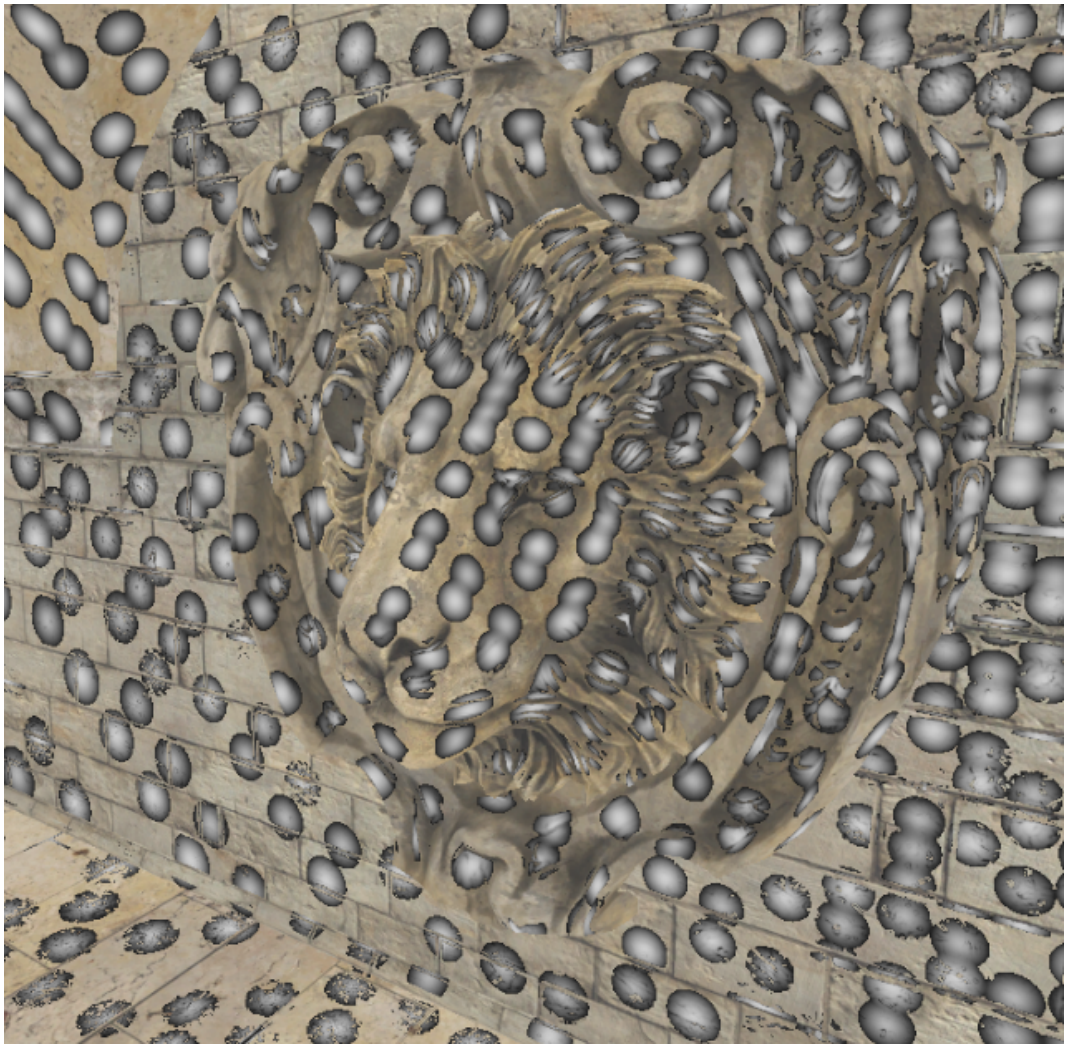

Figure 4.5 | **Surfels.** Sample for surfels, disks with white center and black rim, in a scene. The scene is a part of the Crytek Sponza[3].

4.5.2 Transparency on Surfels

Representing transparency on global illumination surfels doesn't have a trivial solution. Brinck et al. [6] implement transparency using light probes and storing their irradiance with spherical harmonics. Spherical harmonics are a representation of light data on the surface of a sphere. Spherical harmonics use sine and cosine modes instead of pixel cubemaps, which has the advantage to be continuous. The principle for how the data is stored is visualized in Figure 4.6.

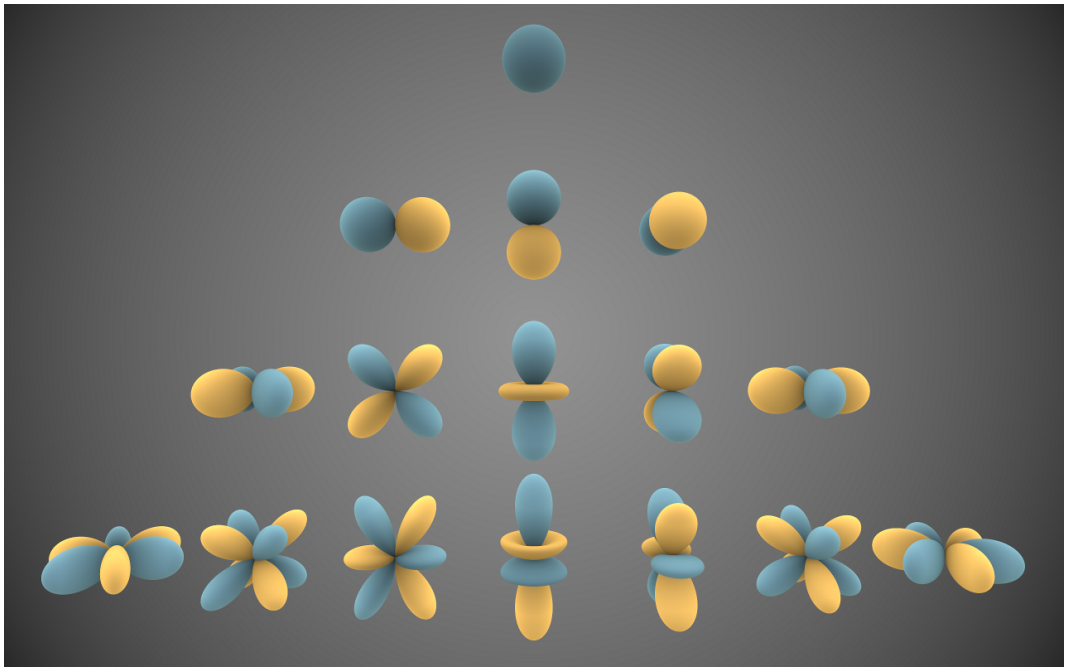

Figure 4.6 | **Spherical Harmonics.** Visual representations of the first 4 bands of real spherical harmonic functions from Inigo Quilez. Blue parts are regions where the function is positive, and yellow parts represent regions where it is negative.

4.6 Gradient-Domain Path Tracing

"Gradient-Domain Path Tracing" was introduced by Kettunen et al. [16]. They use finite differences and path shifts to calculate pixel space gradients. These gradients then are used to reconstruct the image using Poisson reconstruction.

Their inspiration for generating gradients is the observation that often the power spectrum of the gradients of an image has less energy than the power spectrum of the value itself. They demonstrate it on an artificially generated image of black lines on a white background. They predict that their scheme produces less variance than sampling the image directly.

Given a noisy image, and a less noisy gradient image for horizontal and vertical directions, a less noisy image can be generated using Poisson reconstruction. The difficult part is to generate these low-noise gradients because the lighting integral is highly discontinuous. This is because of the geometry term: solid matter in empty space has a density discontinuity on the surface. Each surface represents such a discontinuity, and we are rendering these. Loosely speaking, when a ray hits a surface, there is no guarantee for a neighboring ray to

hit that surface as well. This process is also explained in detail by H. Igehy [13], and they discuss texture filtering as well.

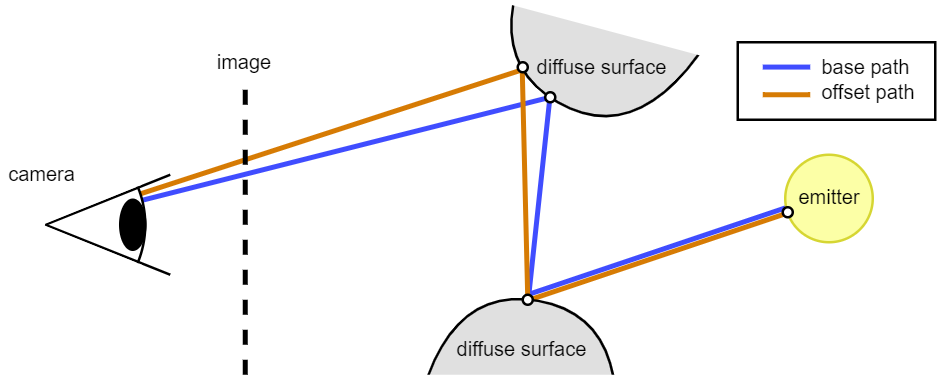

Figure 4.7 | **Base and offset path in gradient-domain path tracing.** The offset path is first traced at a slight angle from the base path. If possible, the path is reconnected with the base path. Such a reconnection is only sensible, if the probably of that modified path is non zero. In this figure, the base path hits two diffuse surfaces consecutively, and there is no obstructions for the offset path, so the paths can be reconnected after two rays from the origin.

The probability of rays hitting different geometry can be reduced by trying to merge their paths as soon as sensibly possible. However, the gradient still needs to be correct, so it cannot violate reflection or refraction laws, and should not have probability zero. A simple example for a base path, and an offset path is shown in Figure 4.7, where they hit two diffuse surfaces until they hit an emitter. Diffuse BSDFs have a non-zero probability for a large section of their hemisphere, so connecting them is possible. In Kettunen et al. [16], they connect offset paths to the primary path only if the current offset point, the current primary point, and the next primary point are diffuse. Reconnecting them still may fail, because the path may be occluded, or the BSDF may be zero in that direction.

4.7 Multiple Importance Sampling

Multiple importance sampling (MIS) is a technique, where multiple sampling strategies are combined [28], because they may have different strengths and weaknesses. For example, sampling refraction patterns using path tracing needs a long time to converge, and point lights practically never converge to the correct solution, because the probability of hitting a point with a random line in space is zero.

In light sampling, light rays travel from the light source. The method tries to find paths from hit surfaces, or the start, to the observer. Light sampling has no difficulties sampling point lights: the opposite is true here: there are fewer paths that transport light, so it should converge faster than a scene with many large area lights.

These samples can be combined linearly, and if using unbiased sampling methods, the linearly combined result is unbiased as well, as long as the sum of sampling weights is 1. They even state that sampling methods in MIS are not required to cover the whole path space, as long as they produce correct results. This means that specialized sampling techniques could be applied to different parts of an image, depending on the material and local lighting circumstances. In path tracing, light rays are sent from the camera into the scene until they find a light source, or we cancel the execution. Their weaknesses, strengths, and the combined result are presented in Figure 4.8.

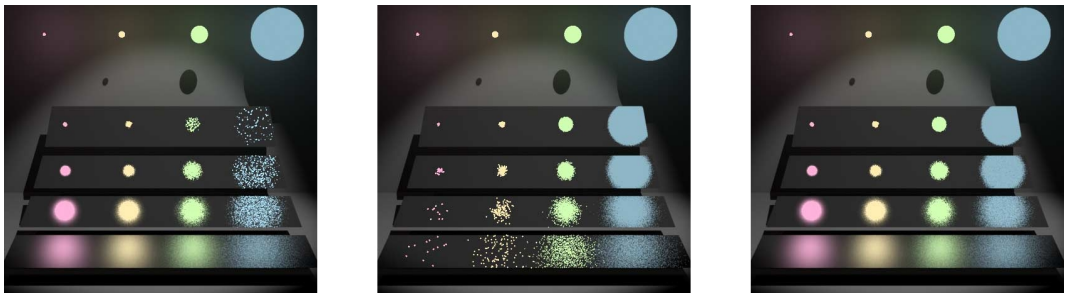

Figure 4.8 | **Multiple Importance Sampling.** Light sampling (left), path tracing (middle) and MIS (right) on a test scene; by [28]. Different sampling methods (left vs middle) produce different variance distributions based on the light conditions. The combined image (right) has much less variance (noise).

4.8 Poisson Image Reconstruction

Poisson reconstruction is an optimization problem, in which approximations of a function and its gradients are given, and the original function shall be reconstructed. Poisson reconstruction is used to merge features into existing images. Instead of directly copying the color, the gradient is applied to the calculated gradient of the source image. This would typically introduce a contradiction because the added gradients are different from the source image gradients. A Poisson solver is used to minimize this error. Such a method is implemented in image editors like Photoshop [11] and Gimp (Heal Operator) [10], [8]. They are called "Healing" and "Seamless Cloning" there.

Poisson reconstruction can be computed with a solver like the Jacobi or the Gauß-Seidel iteration. Practically, an image kernel is applied iteratively on the image until the error is small enough for the application. This kernel has the task of reducing the error, so to solve the problem, the image value is calculated that would have resulted in error $= 0$. The updated color value is this locally error-free value linearly mixed with the previous color. This mixing factor is also called the relaxation parameter. If the factor is not one, the method is called "successive overrelaxation". Negative factors make no sense, as they would worsen the result. Factors larger than two might not converge.

4.9 Thesis Scope

In our master thesis, we implement surfel-based global illumination in the Unity game engine. We discuss surfel count, surfel distribution, and weighting functions. We measure the surfel rendering performance based on instanced and procedural rendering.

We implement poisson reconstruction in Unity on top of the surfel-based global illumination. Although we wanted to use the original tracing method from Kettunen et al. [16], we failed to translate it correctly to simple precision floats and HLSL in Unity, so we kept our much simpler path tracer. The importance of matching finite-difference-definitions is shown.

Two methods for transparency on surfels are proposed, that we haven't found in the literature yet. We implement one of them.

5 Implementation Details

This chapter is about design decisions and implementation details for simplicity and performance. It also discusses surfel distribution, weighting schemes, and an optional preparation step in poisson reconstruction.

5.1 Casting Rays

Path tracing based global illumination requires a method to cast rays. Given a start position and direction, the first intersection with scene geometry is searched. At this intersection, different properties need to be computable, like geometry normals, surface BSDF, and surface color. Both ray tracing and ray marching are viable approaches. We use ray tracing for our experiments.

5.2 Stability

In real time raytracing, we assume that the light information is relatively stable and continuous over world space and time. This has the advantage of having more compute resources to compute the light integrals, but it also introduces incorrectness. It naturally only holds true in a few cases, like calm strategy games, not in action-packed shooters.

When we assume spatial stability, fine details might be lost. For example, hard shadows from light sources with small solid angle can no longer be accurately resolved. This problem has been illustrated in Figure 5.1. The path traced image shows clear, hard shadows, while the surfel-based approach bleeds over the edges. Hard shadows can be generated by light sources with small solid angle.

The same is valid for temporal stability. Changing the intensity of lights is an issue, as this violates our assumption of temporal stability. Examples for this are lightning or flickering lights, but it also happens with moving lights. There the resulting error is sometimes referred

Figure 5.1 | **Spatial Locality.** Lighting of a lion statue in the Crytek Sponza[1] from a small light source. Per-pixel path tracing (left), the same but with blurred global illumination (middle) and surfel-based path tracing (right). The path traced image converged much slower than the surfel approach. The surfel-based approach stayed blotchy however.

to as ghosting, see Figure 5.2. This is an artifact of temporal algorithms like temporal antialiasing, that is hard to avoid except by giving up on temporal stability.

Our surfels are a way of describing/assuming spatial stability, because they are typically much larger than a single pixel. We keep them for multiple frames, which assumes temporal stability.

5.3 Surfel distribution

In our thesis, surfels carry global illumination information. These surfels should be distributed well to balance calculation efforts to where they are most effective at reducing noise while representing global illumination well. Therefore, wherever global illumination changes spatially, a transition zone with high surfel density is needed. Ideally, they would be distributed according to global illumination changes. If the surfels store gradients as well, high densities are only needed where the gradient changes. However, our idea is to keep them stationary for temporal stability and temporal accumulation. Because of that, they are placed in world space and kept for many frames.

This temporal and spatial interpolation introduces errors against the per-pixel path traced baseline. Ideally, a surfel would be needed for every pixel. The premise of our method is that global illumination is relatively consistent in world space, so fewer surfels are used. Finally, if working with a skybox, surfels would need to be placed at infinity or sky height. However,

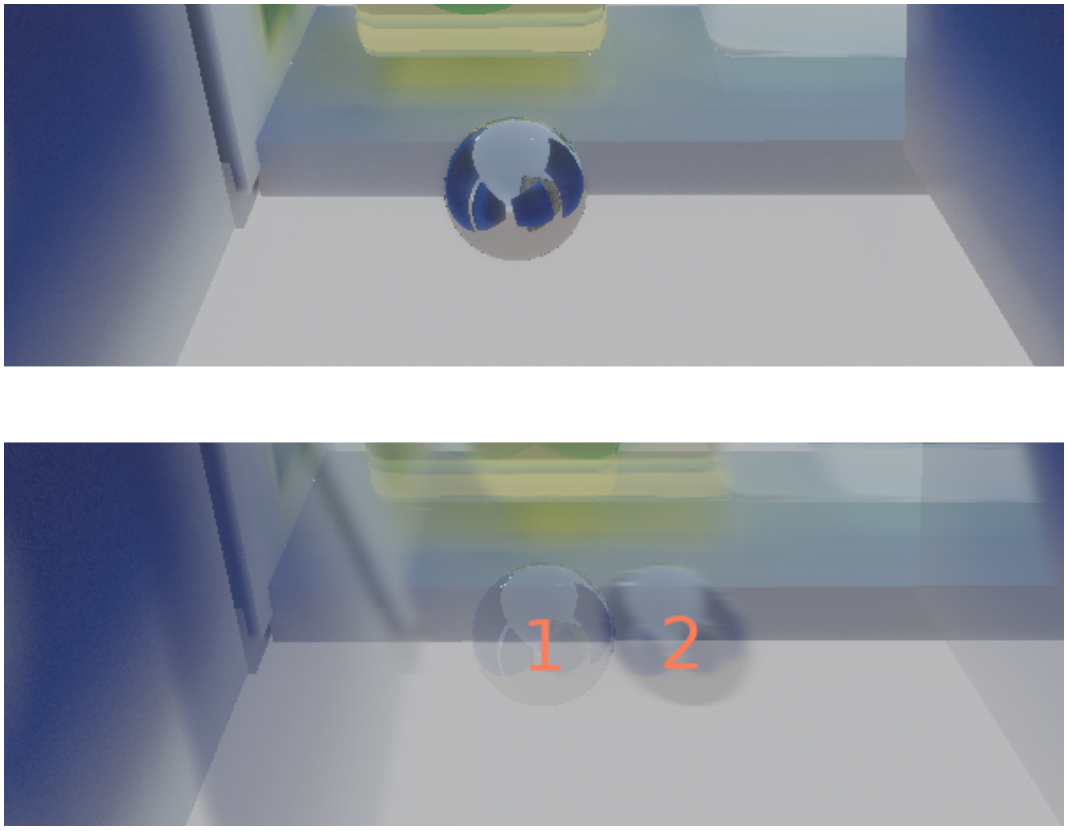

Figure 5.2 | **Ghosting on a transparent surface.** Path tracing results were kept in a moving scene using motion vectors to create this image. The formula for invalidating global illumination on specular surfaces by angular changes was not strict enough. The top image shows the results before movement. The bottom image shows the results after moving the camera to the side. A "ghost" sphere (marked with a two) is clearly visible next to the original sphere (marked with a one).

the sky can be defined as a pure emitter such that its color completely defines it, so we don't need surfels on it.

5.3.1 Initial Distribution

When surfels are spawned for the first time, they are distributed evenly in camera space. They are placed on a Fibonacci sphere, see M. Roberts [24]. Raytracing performs better if close rays are processed in close compute nodes, so the memory needs to be fetched only once. Therefore, ideally, a distribution should be chosen that has this property, e.g., a space-filling curve (like a Hilbert curve) on a sphere. Such a distribution has been described by Purser

et al. [22]. A Hilbert curve is presented in Figure 5.3 Using the Hilbert curve instead of the Fibonacci sphere probably isn't that impactful, because the order will be mixed randomly over the runtime of the game, because the camera will most likely be moving. This could be solved by sorting the surfels according to a such a Hilbert-index-mapping every frame.

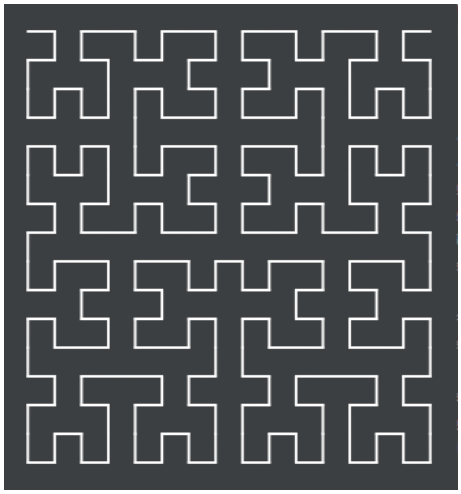

Figure 5.3 | **Hilbert Curve.** A Hilbert curve with 256 vertices, so three subdivisions from the base shape.

To show the locality effect of a space-filling curve like the Hilbert curve, we implemented an easier, but also non-uniform mapping. This mapping first uses the scalar index to find a position on the 2D plane using a Hilbert curve[2], and then transforms this 2D vector into a point on the unit sphere using an octahedron mapping[3]. Such a mapping has been developed to efficiently store normals.

In Figure 5.4, the surfels are drawn, and their color is set by their index modulo 255. The transform using the Hilbert curve is less noisy and thus should provide a bit better performance, because of much better memory locality for the primary rays.

5.3.2 Distribution Updates

Andersson and Barré-Brisebois [1] and Barré-Brisebois et al. [2], describe a method, where the summed weight over all surfels is first rendered to a framebuffer from the perspective of the camera. Then, for each 16x16 tile, a compute shader places a new surfel where the weight is the lowest within that tile if it is below a certain threshold.

[2] http://bit-player.org/2013/mapping-the-hilbert-curve
[3] https://knarkowicz.wordpress.com/2014/04/16/octahedron-normal-vector-encoding/

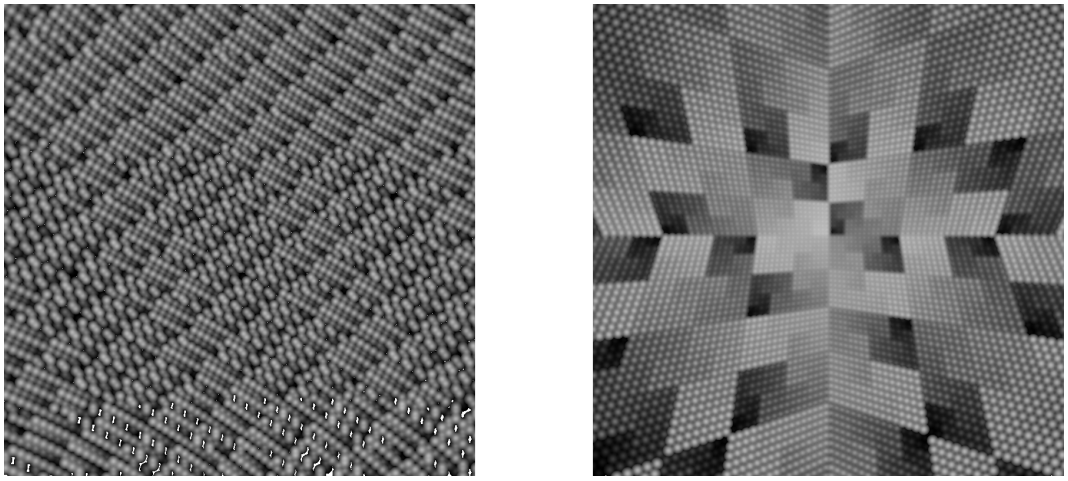

Figure 5.4 | **Spatial Locality for initial surfel distributions.** The distributions were generated from a Fibonacci sphere (left) and a Hilbert curve (right). Their color is set by their index modulo 255 for visualization. The white spots in the left image are pixels where no surfel was drawn. These must be filled in by the distribution updates.

Keeping track of memory and allocations is difficult in a massively parallel environment like GPUs. In Barré-Brisebois et al. [2], an atomic counter is used on a total of 250,000 surfels in a global pool. In our implementation, a similar solution is used: a random range of surfels is selected (16 in our implementation). From this range, the least useful surfel is chosen. Their usefulness is estimated by their alignment with the camera, such that surfels behind the camera are less important. This factor then is multiplied by the number of paths traced for that surfel, so surfels that have had a lot of computation cost spent are not discarded. The idea of this is that the new surfel have a higher value than the destroyed one. This method works surprisingly well, as long as there are enough surfels. If there are too few surfels in the global pool, surfels jump around, and add temporal instability.

In a second pass, the weights of each surfel are updated. If the surfel size is changed by more than a certain factor, e.g., 1.41, the surfel illumination is discarded completely. When the illumination depends on the angle of the viewer, like in a mirror, the weight is reduced on angle change as well.

The distribution should be relatively evenly, so it can be improved by moving the surfels against the gradient of the surfels' weight field minus the weight by the currently inspected surfel. We call this process "Surfel migration". On a continuous field, the gradient of the weight at the position of the surfel would be zero if the weights were independent of surface

properties. Therefore, as an approximation, the rasterized, accumulated weights can be used. Similarly to how the certainty of surfels has to be reduced when their size is changed, it also has to be reduced when they are moved. This simple approximation only works well on surfaces, which are relatively aligned with the image plane. In its current implementation, there are also issues with it on the border of the visible region: Surfels are constantly moving toward the center and being respawned. Because of these issues, we recommend turning "Surfel migration" off in our implementation.

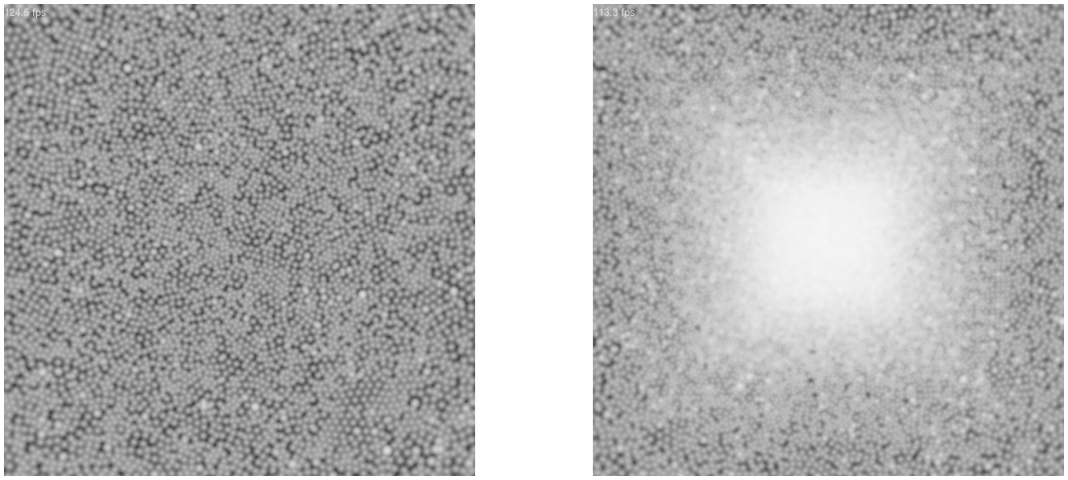

Figure 5.5 | **Surfel density issues when moving the camera.** Surfel distributions before (left), and after (right) moving away from a planar surface using our algorithm. "Surfel migration" was disabled.

When moving away from a surface, while looking at it, a relatively high density of surfels remains, see Figure 5.5. While this improves visual quality in that area, it would be better if the surfels were used in other places for a more even distribution. In future implementations, surfels could be despawned where the density is too high.

5.4 Drawing Projected Surfels

As described in Section 4.5.1, surfels need to be correctly drawn as if they were disks in 3D space. The result of such a projection has been visualized in Figure 5.6. For our implementation, the surfels are drawn as 3D cuboid meshes. These meshes can be defined using their vertices as $[-1, 1]^3$. First, they are scaled to the size of the surfel such that the surfel is fully contained within the cube. They then are rotated such that their original up direction is along the normal of the surface, and positioned in world space.

This transformation is formulated in (5.1). Let $\mathbf{H}$ be the homogenous coordinates of the vertex, let $\mathbf{P}$ be the camera's projection matrix, and let $\mathbf{M}$ be the camera's model matrix relative to the camera. $\vec{c}$ is the respective vertex coordinates as a vector. $\mathbf{T}$ is a translation matrix describing the position of the surfel relative to the camera. $\mathbf{R}$ is a rotation matrix using the rotation of the surfel. $\mathbf{S}$ is a scale matrix, which describes the scaling of the cube to a cuboid matching the size of the surfel.

$$\begin{aligned} \mathbf{H} &= \mathbf{P} \times \mathbf{M} \times \vec{c} \\ \mathbf{M} &= \mathbf{T} \times \mathbf{R} \times \mathbf{S} \end{aligned} \tag{5.1}$$

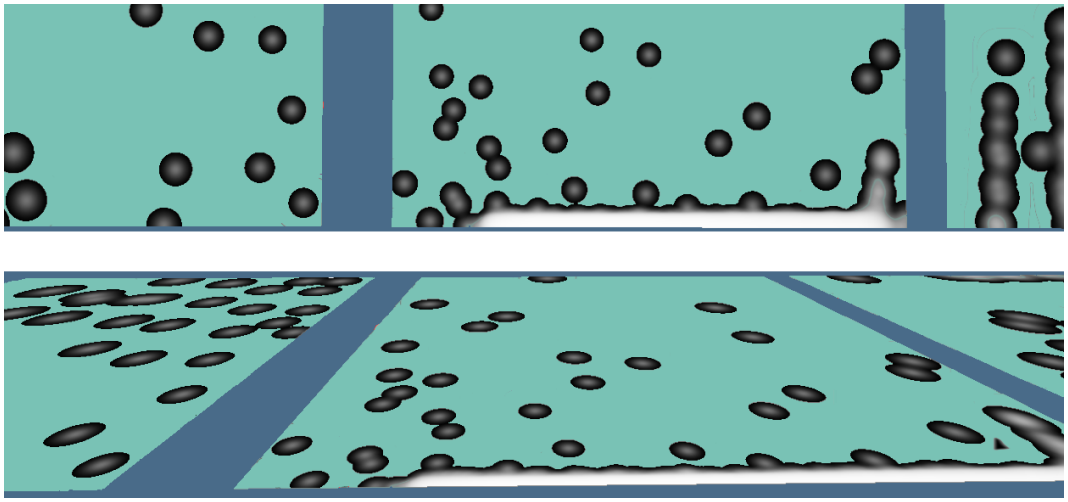

Figure 5.6 | **A few, sparse surfels (black disks with gray center) drawn over three planes at a high (top) and steep (bottom) angle.**

After the surfel geometry has been calculated, a fragment shader is applied, that computes the contribution to the respective pixels in local space. The contribution is calculated using the formula discussed in the next section.

5.5 Surfel Weighting

When illuminating the scene using surfels, each pixel needs illumination data. Often, there are multiple surfels in the proximity of the pixel, so the data has to be interpolated. We use linear interpolation, where weights called $w(s, p)$ are the importance of each surfel s to that

pixel p. This interpolation is formulated by Equation (5.2), there c_p and c_s are the global illumination of the pixel and the surfel respectively.

$$c_p = \sum_{surfels} c_s \cdot w(s,p) \tag{5.2}$$

Neighboring pixels might have different BSDFs, and different distances from the camera. In both cases, surfels must only light pixels, which are close in world space. There are two approaches, that are used to tackle the BSDF issue: Firstly, surfels could store light information about the place, not the material, such that pixels then evaluate their BSDF based on surfel data. Second, as it has been implemented, surfels store illumination information for a specific BSDF, and are specific to the direction of the surface.

Our surfels store specular and roughness values, which are then compared for similarity with the target pixels. Their weighting function is defined in (5.3). Let w be the weight, w_d, w_n, w_r and w_s be the terms for distance, normal, roughness, and specularity. d is the squared distance from the surfel center in local coordinates. N_s and N_p are the normals of the surfel, and pixel respectively. A_s and A_p are the smoothness of the surfel and the pixel. S_s and S_p are the specularity values of the surfel and the pixel.

$$\begin{aligned}
w &= w_d w_n w_r w_s \\
w_d &= \textit{clamp}(1/(1+20\cdot d) - 1/6) \\
w_n &= \textit{clamp}(10\cdot(N_s \cdot N_p) - 9) \\
w_r &= \textit{clamp}(1 - 20\cdot \textit{abs}(A_s - A_p)\cdot \textit{max}(A_s, A_p)) \\
w_s &= \textit{clamp}(1 - 20\cdot \textit{abs}(S_s - S_p)),\ \textit{with} \\
\textit{clamp}(x) &= \begin{cases} 0, & \text{for } x < 0 \\ x, & \text{for } 0 \le x \le 1 \\ 1, & \text{for } 1 < x \end{cases}
\end{aligned} \tag{5.3}$$

This function introduces errors compared to the per-pixel path traced baseline because it is just an approximation. A strict, unbiased function would defeat the purpose of our surfels, as it would need at least as many surfels as there are pixels. Also, a clamping function is used, but a function with a falloff toward zero could be better. Such a falloff could reduce the number of required surfels for regions, where the material properties differ a lot or are noisy.

The goal for this function is to assign different surfels to different BSDFs. If the difference of the BSDFs is small, using multiple surfels may be visually not differentiable. The results of our weighting function are presented in Figure 5.7 and Figure 5.8. Without normal-dependent term, normal maps introduce noise, because some surfels are on slopes, while others may not. Their results would be mixed only by proximity, which is incorrect if the surfels are larger than the details on the normal map.

With normal-dependent term, even a small number of surfels can capture the light effects, that would be expected of a normal map: edges toward the sun are brighter than toward the floor. The cuboid mesh in the second image has a chess masking texture for roughness and metallic, which causes different BSDFs to be used. Weighting surfels with respect to their specific BSDF separates the regions much better. The reflection on the smooth, metallic surface is blurry nevertheless, because a small number of surfels was used to show the effect better, and because direct, sharp reflections are a weakness of representing global illumination using surfels.

Figure 5.7 | **Weighting by normals.** Global illumination on normal-mapped surfels using a weighting function without (middle) or with (right) normal specific terms. The left image is the per-pixel path traced ground truth. The background is a wall from the Crytek Sponza[4].

When calculating the gradient of the surfel emissions on a per-pixel-level, the weighting functions have to be properly handled as well. We use the functions “ddx” and “ddy”, which calculate the gradient based on neighbor pixels, and are available by default in HLSL and GLSL. However, the results were of much lower quality than needed. Probably because of the many discontinuities. The issue was solved by calculating the finite difference for each pixel ourselves. This probably is a bit slower, because we have to evaluate the weight function five times (center, left, top, right, bottom), but it’s required for the poisson reconstruction. The difference in quality can be seen in Figure 5.9: there is much less noise.

A remaining issue is weighting the gradients. In theory, each side of an edge should contribute

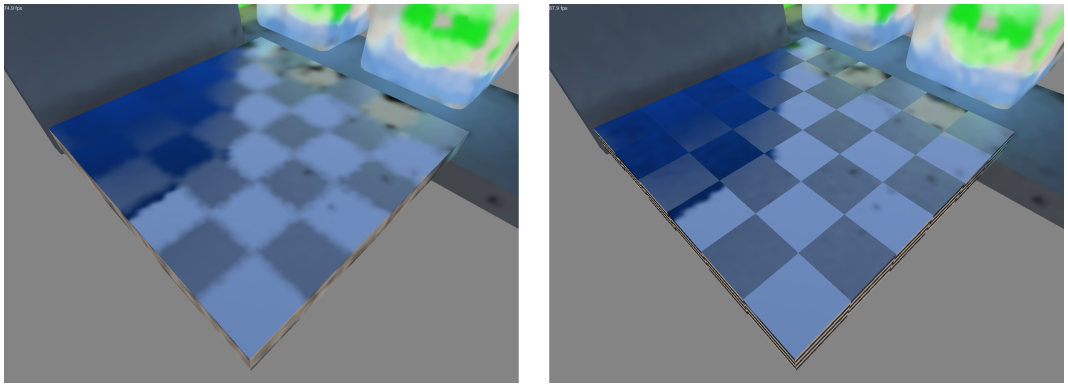

Figure 5.8 | **Weighting by roughness and specularity.** Global illumination on surfels using a weighting function without (left) or with (right) BSDF-specific terms.

exactly once, or the same amount for all. However, using our approach of drawing the surfels procedurally cannot ensure that. If the weight is set to 1, regions with a lot of surfels contribute the most. The issue is especially apparent in Figure 5.10, where the gradients result in discoloring of surfaces. If the weight is proportional to the local weights, small, one pixel thick edges also have aliasing issues. Those issues result in edges of a gradient in a single direction, which result in a high-frequency pattern after poisson reconstruction. That issue is presented in Figure 5.11.

5.6 Pixels per Surfel

Only pixels with a similar BSDF to that of where the surfel was spawned, have a non-zero weight. Therefore, the number of pixels per surfel depends on the weighting function as defined in Section 5.5. The amount depends on a parameter called "Surfel Density". The surfels are spawned with a size s described in Equation (5.4). D is the "Surfel Density", d is the distance from the camera to the surfel, c is the total number of surfels, a is the dot product between the surfel normal and the direction from the camera to the surfel. The term with the dot product makes surfels at an angle larger than those parallel to the viewing plane. Without this term, walls, which are nearly perpendicular to the viewing plane, would need a lot of surfels.

$$s = D \cdot d \cdot 4/(\sqrt{c} \cdot (|a| + 1)/2) \tag{5.4}$$

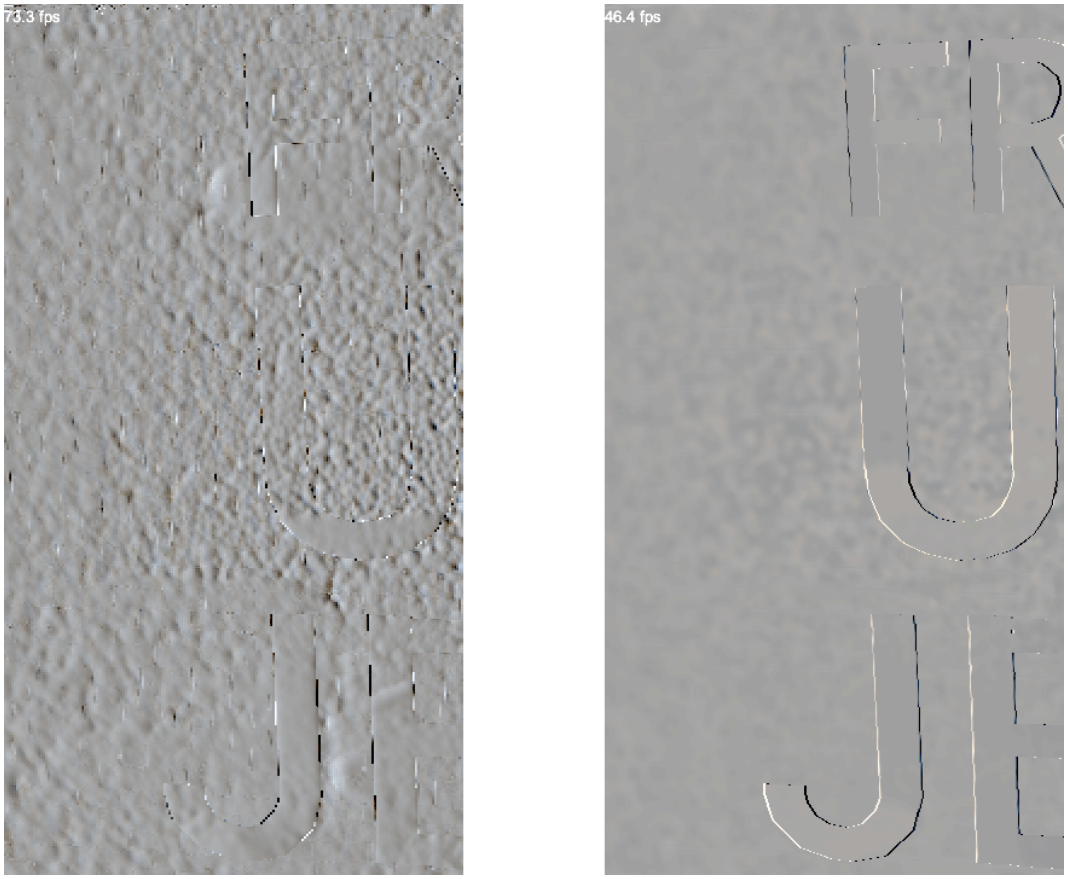

Figure 5.9 | **Comparison of "ddx" and finite differences.** Difference in finite difference along the x-axis of the function ddx (left) and a manually calculated difference of the two neighbor pixels right and left.

5.7 Transparency with Surfels

Transparency like in Section 4.5.2 might work well, but is also complex to implement and out of scope for this thesis. We found two more possible ways, of which we implemented the latter. We have not found a description of either of these methods in the existing literature.

5.7.1 Dithered Surfels

The same way as with deferred rendering in games like "Grand Theft Auto V" as described by A. Courrèges [7], the GBuffer could have its data dithered, when dealing with transparency. Such dithering is shown in Figure 5.12. This only needs a change in the GBuffer generation,

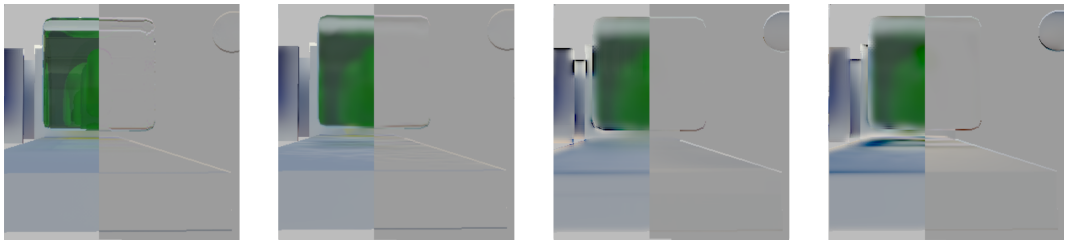

Figure 5.10 | **Problems with weighting gradients.** Image and gradient along the y-axis (right half). The left image is a per-pixel path traced baseline. The middle left image is the same, just rendered with surfels. The gradient there was calculated from the rasterized color values. The middle right image uses the same weight for each surfel to calculate gradients from surfel gradient data. The very right image uses a dynamic weight proportional to the surfel weights (maximum of top, center, and bottom pixel) for the surfel gradient data. The images with surfels look blurred, but that is to be expected, as they cover multiple pixels, and the surface is transparent. The issue is strong tinting along edges toward white and black. This tinting is caused by gradients with too large magnitude.

and a stochastic term in ray casting, but introduces new issues. The dithering might be obvious and be visible as shimmering. Transparencies are limited to a few discrete values, because there isn't any continuous transparency. For a 2x2 dithered region, this would be 0%, 25%, 50%, 75% and 100%. Transparency is faked by making the object discretely invisible at certain pixels. Transparent objects behind other transparent objects (of the same or less opaqueness) need a different dithering pattern to be visible.

5.7.2 Pass-through Surfels

As an alternative, surfels could not only store the color values of the first hit surface, but also of those seen behind it. This has the benefit that the range of transparency isn't visibly noticeable and that there is no dithering pattern. Even if the diffuse part of the transparent surface itself may be rough and may not change its appearance to the camera angle, the background can change by changing the camera angle, so it has to be updated similarly to the worst case: a highly reflective mirror. This sensitivity could be reduced when the material is only slightly transparent (e.g., 90% opacity). In our implementation, pass-through transparency is used. We currently don't change the sensitivity when the surface is only slightly transparent, as we don't get that information available from the GBuffers. Unity's built-in GBuffer is drawn with opaque surfaces only, and we would need to implement a custom GBuffer pass to get transparency information. For the BSDFs, we define that the

Figure 5.11 | **Aliasing issues by proportional weights.** The left side of the image is the poisson reconstructed global illumination. The right side shows the calculated gradients along the y-axis. The image shows letter with depth, where the top side of the letters is barely visible. This one pixel thick edge produces incorrect gradients, where it is rasterized into pixels. These gradients are then transformed into a high-frequency pattern (similar to ringing) by poisson reconstruction.

outgoing direction (see Section 4.3.1) might be on the back side of the surface as well. This allows us to model transparent and translucent materials like glass without additional functions.

Figure 5.12 | **Dithered tree leaves in Grand Theft Auto V.** The image was created by A. Courrèges [7].

5.8 Secondary and tertiary bounces

In path tracing, the process of tracing a path is an iterative process of tracing rays until an emissive surface has been hit. When no emissive surfaces are found, the process would not terminate, so a limit has to be introduced. In our implementation, this is a variable, that can be set dynamically at runtime. When the limit is reached, the algorithm assumes that light cannot be reached via this path.

When a ray hits a surface, the surface is not emissive, and the ray is emitted from the surface based on the BSDF, this is called a "bounce". The number of bounces is therefore the same as the number of rays on the path minus one. When starting the thesis, we were not sure how many ray bounces are required. When working with specular objects like mirrors or glass, the amount needed can be pretty high to get accurate appearances. Especially for glass, they are responsible for total inner reflections, which are an important part for the look of glass.

In mirrors, the reflected scene would look too different from the normal scene, as the reflected scene has a budget with one less bounce. Multiple bounces are needed too, if emissive surfaces cannot reach the surface directly, e.g., if it is obstructed by buildings, cave walls, or similar. The effect of the number of bounces is demonstrated in Figure 5.13.

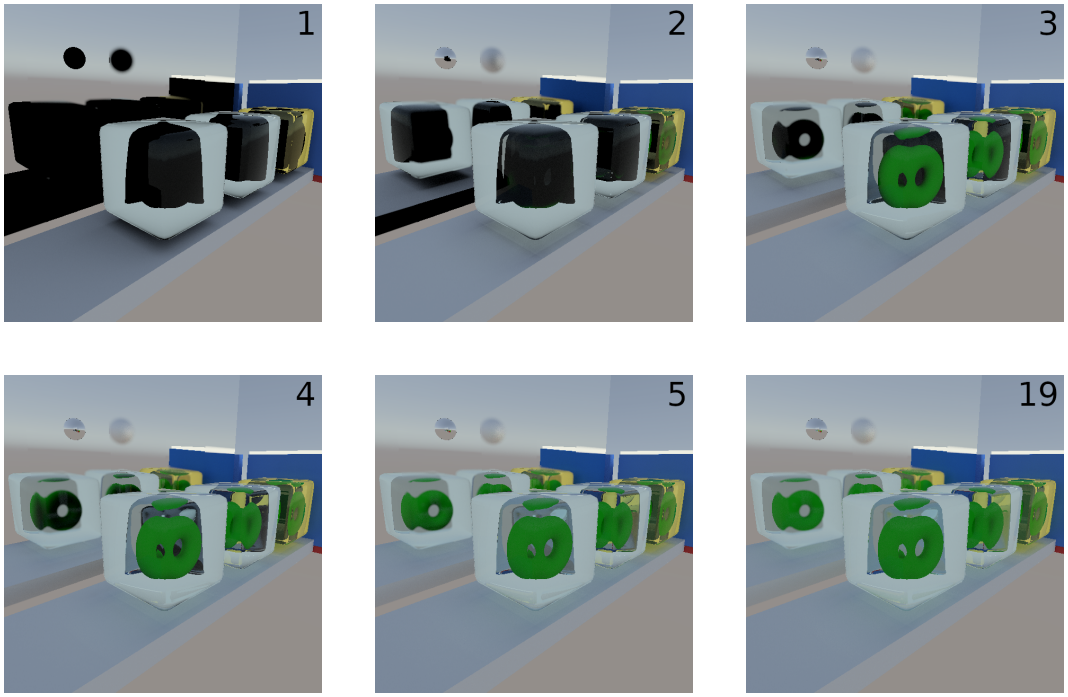

Figure 5.13 | **Maximum number of bounces, and effect of this limit.** The number in the upper right corner denotes the limit.

We recommend to set the maximum number of bounces to a much higher value than our initial guess, e.g., five. This limit could be set lower, if the scene is well-lit, and there are only diffuse materials.

5.9 Drawing Surfels in Unity

We implemented two ways to draw surfels efficiently in Unity. The surfel drawing process is like drawing lights in deferred rendering: there are 3D shapes in world space, that add their contribution to all pixels inside (3D) that shape with some weight. The easiest, but also the slowest version would be to create a GameObject for every surfel. The first approach, we implemented, is to draw the surfels in a series of instanced draw calls. Unity requires the user to define an array of matrices for the transforms of the instanced-drawn meshes. Since the surfels are kept exclusively in GPU memory, the data would have to be transferred from the GPU to the CPU and to the GPU again. Instead, the position is calculated in the vertex shader, so the positions don't need to be transferred. Because of this, all matrices on the CPU side are unit matrices, and contain no actual data. Unity also limits the number

of drawn surfels to 1023 per draw call[5], so multiple draw calls have to be issued. When drawing, this special case of unit matrices is not optimized in Unity, and therefore Unity uploads a matrix for each surfel at each frame (potentially multiple times).

The second way is to draw the surfels procedurally. There, Unity doesn't enforce a matrix array. The performance comparison of these two methods can be found in Section 6.2.

5.10 Poisson Reconstruction

Solving the Poisson problem using the Gauß-Seidel iteration is a repeated application of a linear kernel with multiple input channels (see Section 4.8). Therefore, for N iterations, there should exist a linear kernel, that has the same effect by just being applied once. While we didn't compute this kernel we found a simple approximation. The ideal kernel might not be separable, and expensive to calculate.

Thinking about what such a kernel does on a blurred image, we found a preparation step that brings a large gain in convergence. A "Signed Gaussian Kernel" is applied in the x/y direction of the finite differences along the same axis. This is effectively a weighted integral, weighted by a Gaussian kernel. It is defined as $f(x) = \textit{signum}(x) * \exp -x^2$. If x is zero, $f(x)$ is defined as zero. This function has been plotted in Figure 5.14. The function is then discretized and divided by the sum of either left or right half (the center is zero). First, the gradients are "blurred" along their axis. Then, those blurred gradients are added to the color data. The main components of this preparation step have been visualized on the Friedrich Schiller University Logo in Figure 5.15.

We analyzed their theoretical convergence behavior by applying Gaussian blur to a ground truth image, and calculating its derivatives. Then the original image was reconstructed using the iterative solver. The "Prepared" method includes our additional step with the signed gaussian. The "Normal" method is solving it without an additional step. In Figure 5.16 and Figure 5.17, their root-mean-square-error is displayed by the number of Poisson iterations taken. The test was executed on an image of the FSU Jena logo of the size 1804 x 604 pixels. It was chosen, because it contains a photograph-like part with mixed-intensity gradients, and text with strong gradients.

When using gradients to reconstruct the image, it is important to use the same scheme for the reconstruction as for the calculation. Typical examples are one-sided finite differences

[5] https://forum.unity.com/threads/gpu-instancing-count-limit.591397/

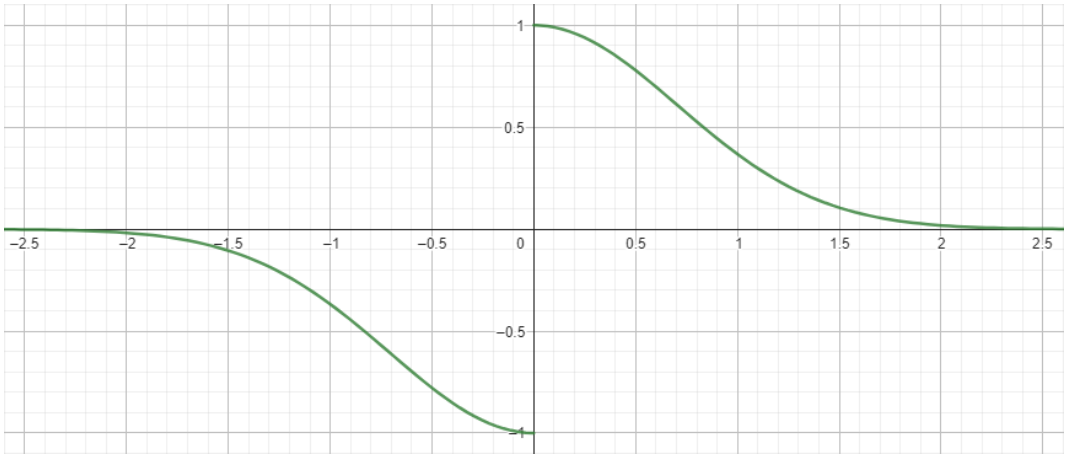

Figure 5.14 | **Signed Gaussian Kernel.** The signed Gaussian kernel function. Without normalization.

Figure 5.15 | **Different parts of the reconstruction process with preparation step.** Upper left: original, middle left: blurred original, lower left: blurred original plus "blurred" gradients, upper right: gradient on x-axis, middle right: "blurred" gradient on x-axis. The letters were reconstructed pretty well by the preparation step along the x- and y-axis.

($[-1, +1]/h$), or the symmetric finite difference ($[-1, 0, +1]/(2h)$, where h is the step size) as we use it. The consequences of using arbitrary or even pseudo-random kernels can be seen in Figure 5.9.

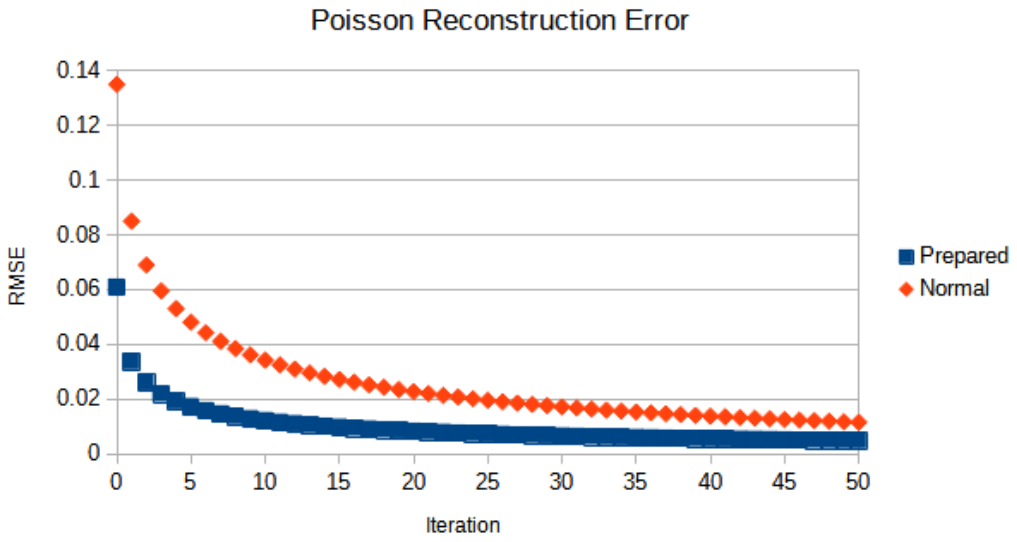

Figure 5.16 | **Convergence comparison on FSU Logo with 6px ≡ 1 sigma Gaussian blur.**

Figure 5.17 | **Convergence comparison on FSU Logo with 20px ≡ 1 sigma Gaussian blur.**

6 Evaluation

6.1 Surfel Count

Choosing the maximum number of surfels is a trade-off like the trade-offs in Section 5.5. More surfels can reproduce fine details better, but they also have a linear rising cost for ray traversal, a cost for being drawn each frame, and for each surfel placement iteration.

The convergence test was executed on the bathroom scene from Kettunen et al. [16]. The results are presented in Table 6.1. The mesh data has been translated from Mitsuba renderer to FBX using a conversion tool in Rem's Engine[1], which outputs GlTF. The GlTF file then was translated to FBX in Blender[2], because Unity does not support GlTF, and Rem's Engine only exports GlTF.

The test was executed at 1280 x 720 pixels like in Kettunen et al. [16], and the camera was placed in a similar spot. For all tests, we waited until the image was converged to the eye, which means there no longer was any visible change. The "Surfel Density" parameter was set to 6, which makes the surfels relatively large, because a lot of surfels are needed in this scene, e.g., to fill the creases in the tiles. When the surfels were relatively stable, we set the minimum surfel weight to zero, which disabled further surfel placement updates.

A noticeable issue of lower surfel counts was that the bars in the windows were not thick enough for two different surfels to be placed. The issue is presented in Figure 6.1. This resulted in their global illumination data being mixed or overridden, and therefore producing wrong results because the glass was brightly lit, and the bars were in shadow from the direct light from outside and above. The sky was enabled for faster convergence. At the highest surfel counts at 720p, it would be more sensible to use pixel space tracing, as the cost is then much higher than without surfels, and that defeats the purpose of surfels.

[1] https://github.com/AntonioNoack/RemsEngine/
[2] https://www.blender.org/

Figure 6.1 | **Varying total number of surfels.** Comparison between pixel-traced result (left), and surfel-based with 65536 (middle) and 4194304 (right) surfels. Notice the partially missing window bars in the image in the middle.

Table 6.1 Root-mean-square-error (RMSE) using surfels for global illumination based on path tracing using a bathroom scene.

Surfel Count	RMSE
65536	0.0698
131072	0.0626
262144	0.5375
524288	0.0490
1048576	0.0430
2097152	0.0403
4194304	0.0390

6.2 Surfel Drawing Performance

Drawing surfels using instanced rendering is a huge bottleneck when drawing lots of surfels, see Table 6.2. It costs 33ms per 1M drawn surfels, which would be an effective bandwidth of about 2 GB/s (assuming a matrix has a size of 64 bytes for 16 single precision entries). However, the tested GPU is connected to the CPU via 16x PCI Express 3.0, so the total bandwidth should be 16 GB/s. So there are more bottlenecks than just PCI Express bandwidth. The solution is to draw the surfels procedurally. There, Unity doesn't enforce a matrix array.

6.3 Convergence

In this section, the convergence of per-pixel path tracing is compared to the convergence of surfel path tracing. The baseline is the per-pixel path tracer, that was used in the previous sections. It is set after the image no longer changes visibly to the eye. The maximum number

Table 6.2 Performance of drawing surfels in Unity

Surfel Count	Instanced FPS	Procedural FPS
262144	53.0	120
524288	27.0	120
1048576	13.5	120
2097152	6.3	76
4194304	3.1	39

This performance measurement was conducted on an Nvidia RTX 3070 (Driver 516.93), AMD Ryzen 5 2600, 32 GB DDR4 memory with Unity 2023.1.0a15. It was running at 600x600 pixels to reduce rasterizer bottlenecks. Surfels were drawn twice per frame.

of rays per path was set to eight. In the experiments, the bathroom scene from Kettunen et al. [16] was used.

The root-mean-square-error (RMSE) metric is used to evaluate the error. It is evaluated after tone mapping the global illumination from linear color space to a clamped linear color space. Unity then converts it to sRGB internally. For tone mapping, a function similar to Reinhard tone mapping is used for simplicity. The formula is described in Equation (6.1), where r, g, and b are the colors of the red, green, and blue color channels. The subscript "in" denotes the linear input values, and subscript "out" denotes the resulting, clamped values. We compare the global illumination instead of the final color, because that is just a color factor. Pixels with lower RGB values just would be "weighted" less.

$$\begin{aligned} k &= \frac{1}{1 + max(r_{in}, g_{in}, b_{in})} \\ r_{out} &= r_{in} * k \\ g_{out} &= g_{in} * k \\ b_{out} &= b_{in} * k \end{aligned} \tag{6.1}$$

6.3.1 Per-Pixel Path Tracing

Figure 6.2 shows the convergence of the baseline per-pixel path tracing compared to using Poisson reconstruction. Using Poisson reconstruction helps convergence at the start, but our method to calculate the gradients is inefficient. This is, because we trace four extra paths

per base-path. Path tracing is the bottleneck of the application. So, if the baseline runs at about 90 fps, the Poisson reconstruction runs at about 18 fps.

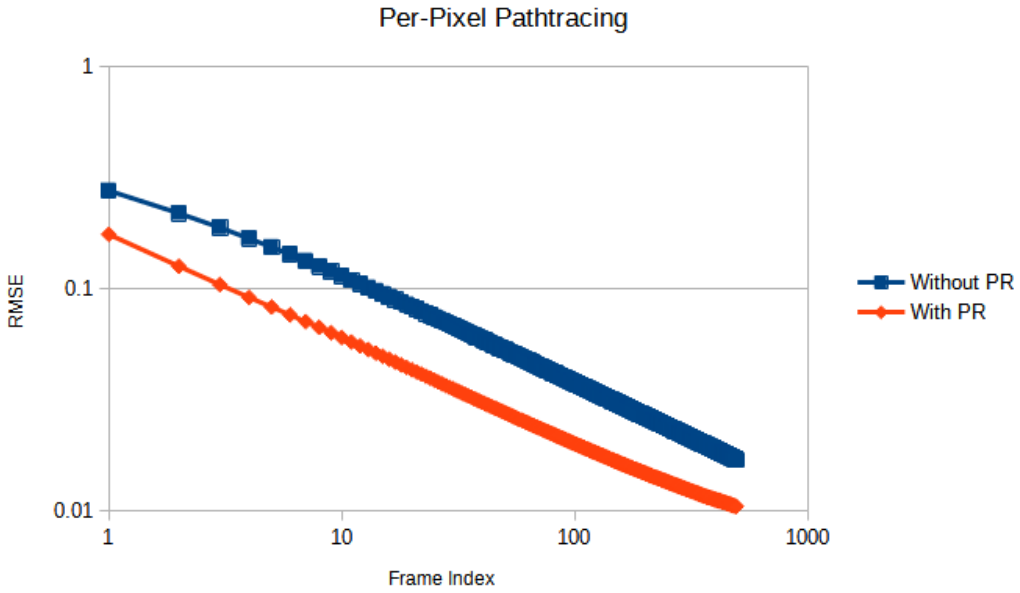

Figure 6.2 | **Per-Pixel path tracing convergence.** "Without PR" is the baseline without Poisson reconstruction, "With PR" uses Poisson reconstruction with 30 iterations, without an additional preparation step.

In the next experiment, the preparation step discussed in Section 5.10 was used. First with a blur-kernel size of 21 pixels and 10 iterations (see Figure 6.3). Then with the same kernel and 30 iterations (see Figure 6.4), because the process was converging much worse. In our tests, the preparation step and blurring only worsen the error on the bathroom scene.

6.3.2 Surfel-based Path Tracing

Since the gradients are incorrect for the surfels, there is a noticeable error against the baseline when Poisson reconstruction is used. The gradients are too small in this scene, and miss details, as has been displayed in Figure 6.5. Since surfels are just an approximation, they do not converge to the baseline either.

All surfel-based experiments were conducted on a surfel pool size of 262144 (2^{18}), because it is a good tradeoff between performance and quality, and a similar value was chosen by Brinck et al. [6]. Other surfel counts are discussed in Section 6.1. For the initial surfel distribution,

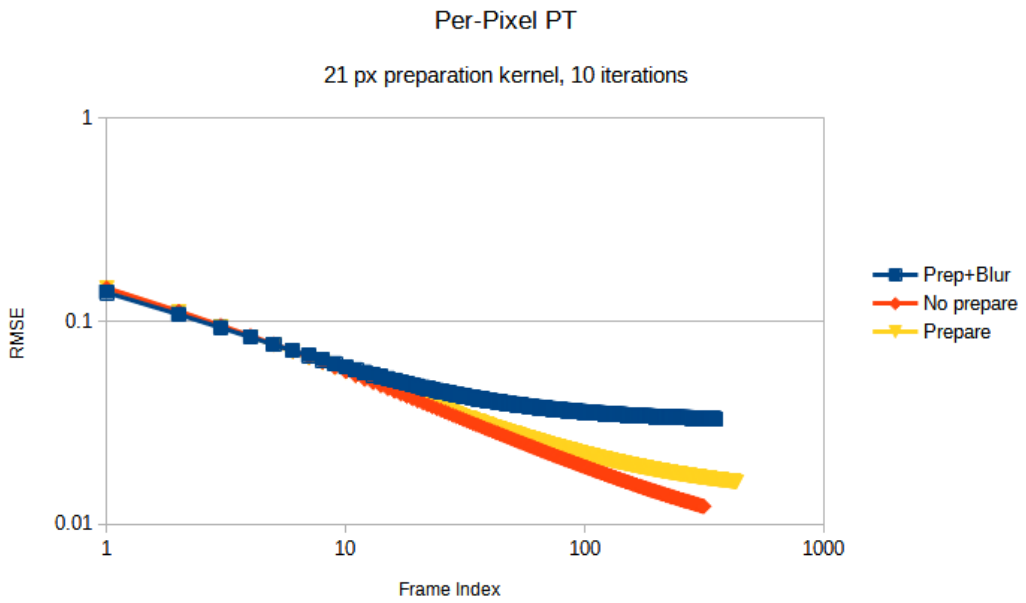

Figure 6.3 | **Poisson Reconstruction convergence, 10 iterations.** "No prepare" is poisson reconstruction without the preparation step, "Prepare" uses it, but does not artificially blur the color information, "Prep+Blur" uses it, and artificially blurs the color information.

the projection with the Hilbert curve was used (see Section 5.3). First, the convergence was measured depending on different surfel density parameters (two, three, and six). The results are presented in Figure 6.6. Larger surfels (higher density) converges faster first, but has a slightly higher error after many frames. The faster convergence at the start can be explained by that more surfels overlap, which reduces the variance. When the illumination is converged, larger surfels have less granularity. Also, higher surfel densities cause the surfels to be temporarily more stable (less despawning) using our surfel distribution algorithm. The test with a surfel density of two had about five to ten constantly unstable surfels, while the experiment with a surfel density of six was stable for all except one to two surfels. Compared to per-pixel path tracing, surfels deliver more stable results first, but later get surpassed by the more accurate method.

For the tests using Poisson reconstruction using the flawed gradients, the extra preparation was disabled, because it converged slower in per-pixel path tracing. The convergence is presented in Figure 6.7. The experiment only converges to about double the RMSE of the experiment without Poisson reconstruction. The more iterations are used on incorrect

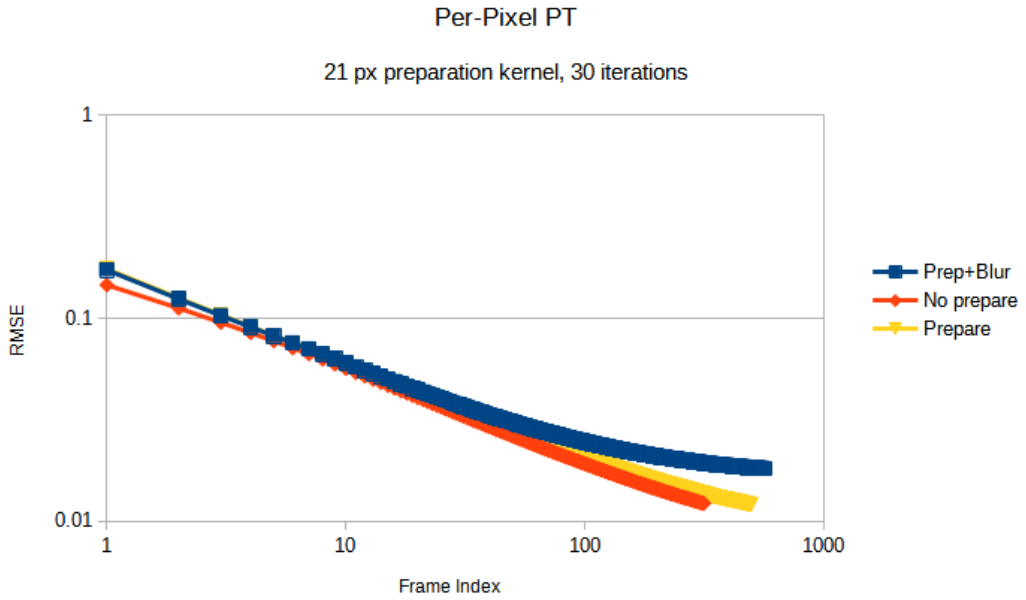

Figure 6.4 | **Poisson Reconstruction convergence, 30 iterations.** "No prepare" is Poisson reconstruction without the preparation step, "Prepare" uses it, but does not artificially blur the color information, "Prep+Blur" uses it, and artificially blurs the color information.

gradients, the worse the results get. This is shown in Figure 6.8, where different numbers of iterations have been chosen while keeping the surfel density (6) and count (262144) the same. The density was chosen to be six because it delivered the fastest convergence for a small number of frames, see Figure 6.6.

In conclusion, our results show that our scheme to calculate gradients is not good enough for Poisson reconstruction on surfels. They also show that the preparation step discussed in Section 4.8 is not useful for our sample scene.

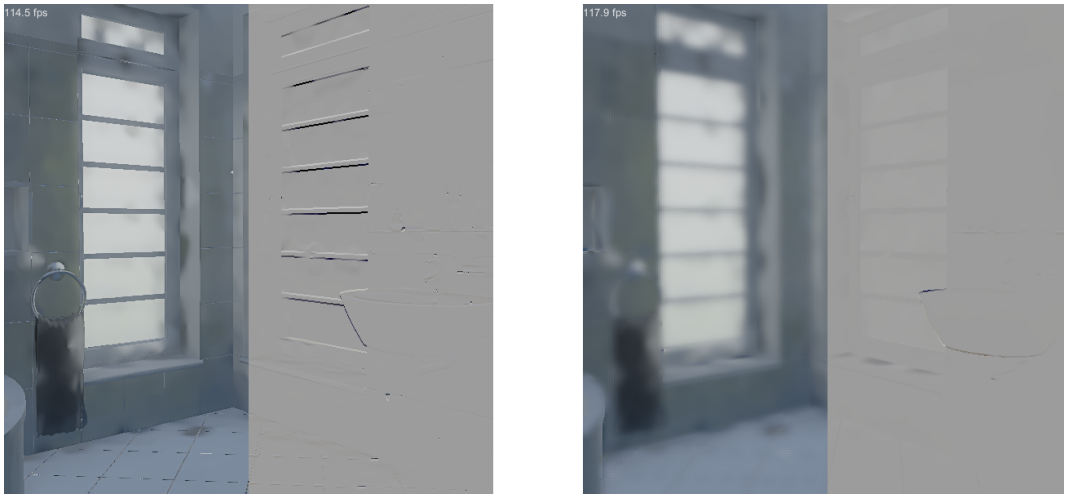

Figure 6.5 | **Gradients missing details and their magnitude being too small.** The image is part of the bath benchmark. The left side of each image is the result after Poisson reconstruction (10 iterations, no preparation step). The left image is the theoretical gradient, just from pixel information. The right side of each image is the gradient along the y-axis. The right image is what has been calculated from the stored gradients within the surfels.

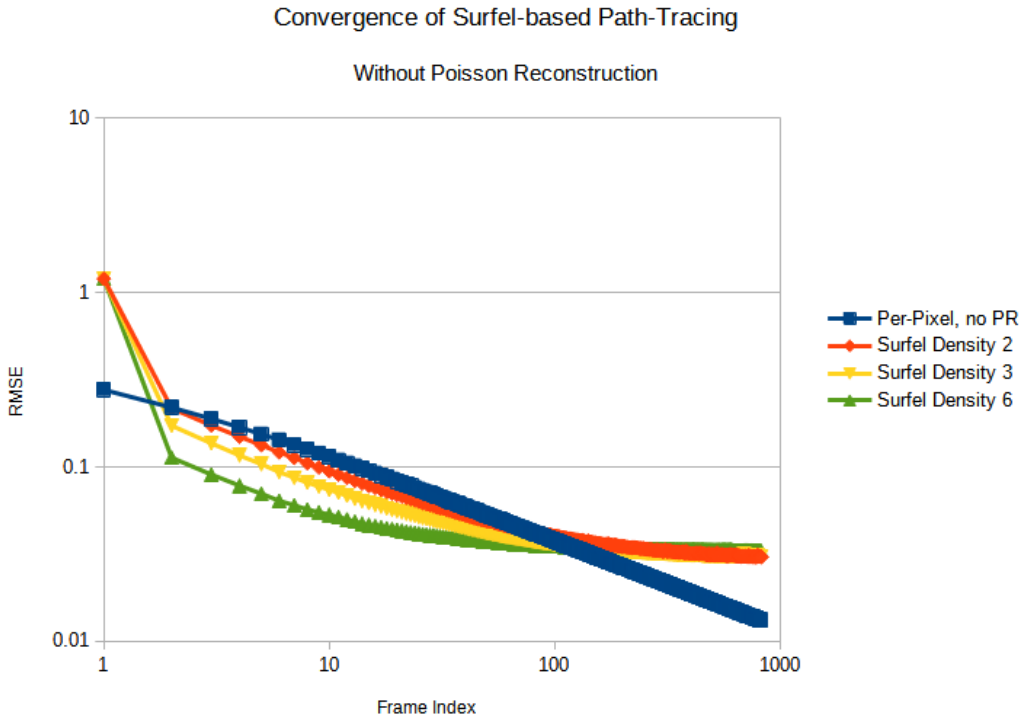

Figure 6.6 | **Convergence of surfel-based path tracing compared to path tracing per-pixel.** "Surfel Density 2/3/6" denotes that the surfel density was set to two, three, and six respectively.

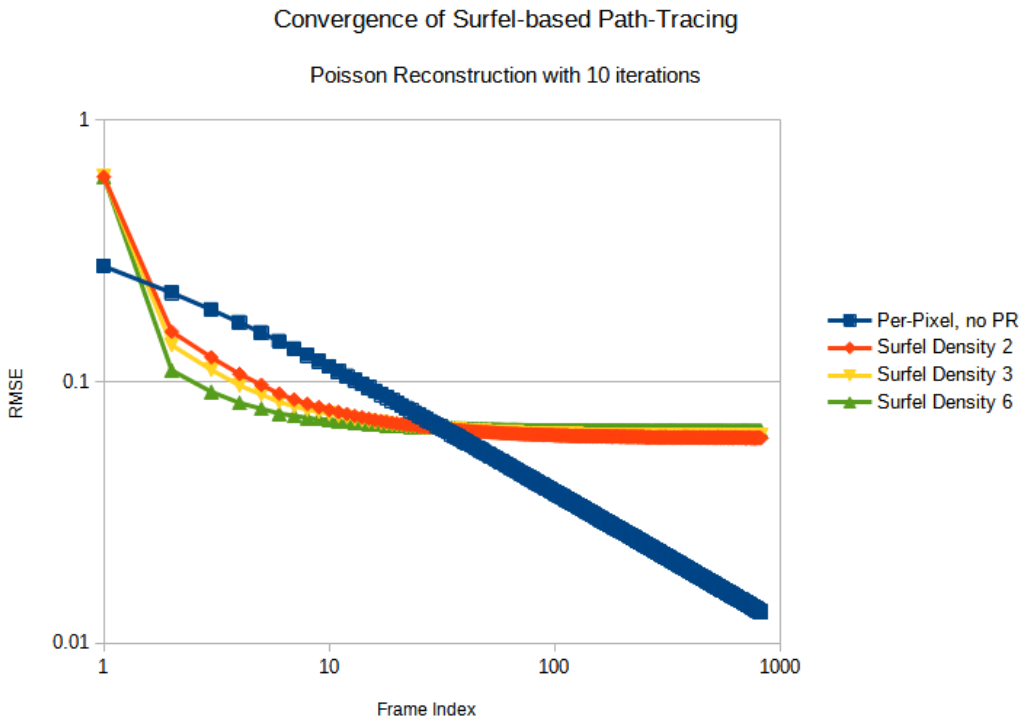

Figure 6.7 | **Convergence of surfel-based path tracing with Poisson reconstruction compared to path tracing per-pixel.** "Surfel Density 2/3/6" denotes that the surfel density was set to two, three, and six respectively.

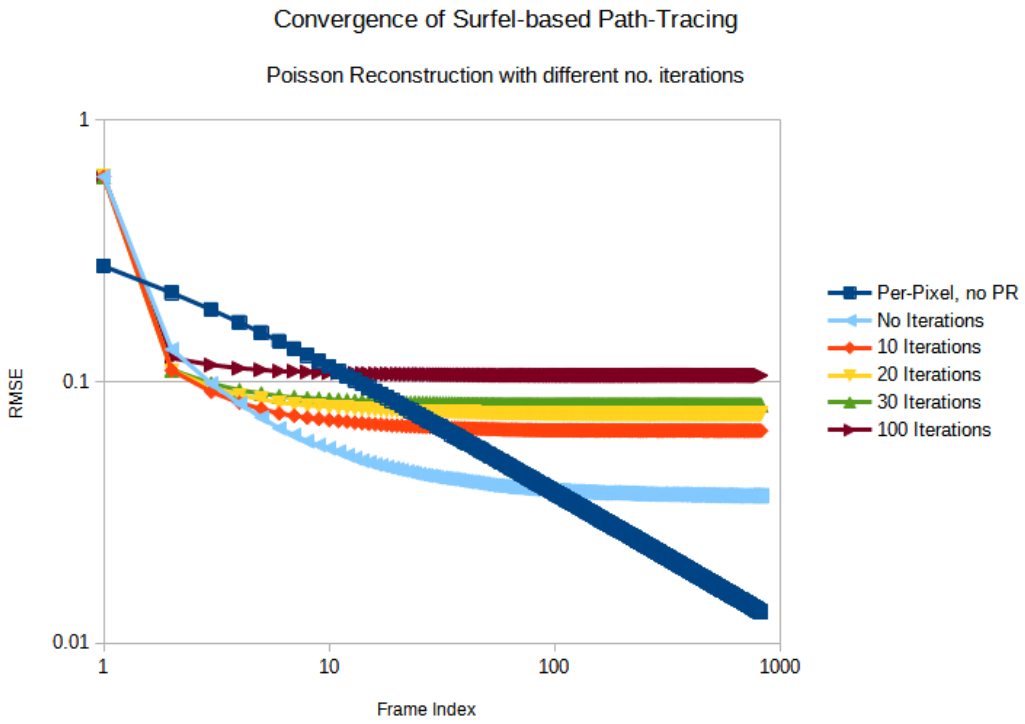

Figure 6.8 | **Convergence of different iteration counts with surfel-based path tracing with Poisson reconstruction.**

7 Future Work

In conclusion, real-time global illumination has been implemented in Unity in this thesis. Our project can be downloaded from `https://github.com/AntonioNoack/Unity-SurfelGI` and reused under the MIT license. Upto two million surfels can be drawn at fluent frame rates (see Section 5.9). These surfels store local global illumination within the scene. The surfels' illumination is calculated using path tracing.

Global illumination reduces noise in path-tracing, when only a few frames are displayed until the image changes. This makes surfels for global illumination useful compared to per-pixel path tracing, even if the scene is quickly changing.

Our method of drawing surfels in Unity could potentially be optimized by calculating the ellipse of a surfel directly in the vertex shader and drawing each surfel as a quad instead of a box. As another optimization, the depth buffer from the GBuffer could be copied to the surfels' depth buffer. This allows pixels to be discarded before entering the fragment shader stage[1]. Another optimization would be deciding where less accuracy than single precision does not matter for storing surfels. This would save bandwidth, which can improve performance for drawing surfels as well.

Calculating the correct gradients in global space should enable working Poisson reconstruction on surfels. Using experiments in Section 6.3, we found that the method presented in this thesis is not good enough for reliable gradients. A compute shader could potentially calculate the gradients for each side of an edge, and then add them together to get a global gradient. This could be implemented by inserting the surfels into a grid, such that the program has data for all nearby surfels. Alternatively, edges could be detected where the weighted gradients would be written to two separate buckets depending on which side of an edge a pixel is on. This would not solve the issue for corners though.

Another aspect of future work is to improve path tracing efficiency. Rays could be grouped by proximity, so they access the same memory for less used bandwidth. Nvidia introduced

[1] https://www.khronos.org/opengl/wiki/Early_Fragment_Test

"Shader Execution Reordering"[2] with their RTX 4000 series of GPUs, which has a similar idea. When Unity supports it, a performance comparison would be interesting. It is a feature that needs to be used by the developer explicitly.

The path tracing efficiency could also be improved by implementing the approaches from Kettunen et al. [16]. Their bidirectional approach poses the difficulty that they use dynamic allocations while path tracing. These are not supported by languages like HLSL and could be implemented using a global pool, or be replaced with static memory.

The light contribution distribution could also be estimated like in Bitterli et al. [4] and Ouyang et al. [20], to sample more effectively. They use statistical reservoirs to share information about light sources with neighboring pixel groups. To apply them to surfels, neighbor surfels would need to be found.

More advanced transparency techniques like in Brinck et al. [6] could be merged with our approach, so they support effects like fog, god rays, and can illuminate quickly moving particles efficiently. This would probably need two types of surfels in the same scene: simple, like ours, and complex with spherical harmonics.

[2] https://developer.nvidia.com/sites/default/files/akamai/gameworks/ser-whitepaper.pdf

8 Bibliography

[1] Johan Andersson and Colin Barré-Brisebois. "Shiny Pixels and Beyond: Rendering Research at SEED". In: *Game Developers Conference* (2018). URL: https://developer.nvidia.com/blog/video-series-shiny-pixels-and-beyond-real-time-raytracing-at-seed/.

[2] Colin Barré-Brisebois et al. "Hybrid Rendering for Real-Time Ray Tracing". In: *Ray Tracing Gems* (2019). DOI: 10.1007/978-1-4842-4427-2_25.

[3] Pablo Bauszat et al. "Guided Image Filtering for Interactive High-quality Global Illumination". In: *Computer Graphics Forum (Proc. of Eurographics Symposium on Rendering EGSR)* 30.4 (June 2011), pp. 1361–1368. DOI: 10.1111/j.1467-8659.2011.01996.x.

[4] Benedikt Bitterli et al. "Spatiotemporal reservoir resampling for real-time raytracing with dynamic direct lighting". In: *ACM Transactions on Graphics* (July 2020). DOI: 10.1145/3386569.3392481.

[5] Mario Botsch and Leif Kobbelt. "High-Quality Point-Based Rendering on Modern GPUs". In: (Nov. 2003). DOI: 10.1109/PCCGA.2003.1238275.

[6] Andreas Brinck et al. "Global Illumination Based on Surfels". In: *Siggraph Advances in Real-Time Rendering in Games* (2021).

[7] Adrian Courrèges. "GTA V - Graphics Study". In: (Nov. 2015). URL: https://www.adriancourreges.com/blog/2015/11/02/gta-v-graphics-study/.

[8] GIMP Documentation. "Gimp Heal Tool". In: *docs.gimp.org* (). URL: https://docs.gimp.org/2.10/en/gimp-tool-heal.html.

[9] Eric Galin et al. "Segment Tracing Using Local Lipschitz Bounds". In: *Computer Graphics Forum* (Mar. 2020). DOI: 10.1111/cgf.13951.

[10] Todor Georgiev. "Image Reconstruction Invariant to Relighting". In: *EG Short Presentations*. Ed. by John Dingliana and Fabio Ganovelli. The Eurographics Association, 2005. DOI: 10.2312/egs.20051024.

[11] Todor Georgiev. "Photoshop Healing Brush: a Tool for Seamless Cloning". In: *Workshop on Applications of Computer Vision in conjunction with ECCV 2004, Prague* (Jan. 2004).

[12] Mathias Holst and Heidrun Schumann. "Surfel-Based Billboard Hierarchies for Fast Rendering of 3D-Objects". In: *Eurographics Symposium on Point-Based Graphics*. Ed. by M. Botsch et al. The Eurographics Association, 2007. ISBN: 978-3-905673-51-7. DOI: 10.2312/SPBG/SPBG07/109-118.

[13] Homan Igehy. "Tracing Ray Differentials". In: *SIGGRAPH '99 Proceedings* (1999). DOI: 10.1145/311535.311555.

[14] Wenzel Jakob. "Mitsuba Documentation, Version 0.5.0". In: (Feb. 2014). URL: https://www.mitsuba-renderer.org/releases/current/documentation.pdf.

[15] Maik Keller et al. "Real-time 3D Reconstruction in Dynamic Scenes using Point-based Fusion". In: *International Conference on 3D Vision - 3DV 2013* (June 2013). ISSN: 1550-6185. DOI: 10.1109/3DV.2013.9.

[16] Markus Kettunen et al. "Gradient-Domain Path Tracing". In: *ACM Transactions on Graphics 34(4) (Proc. SIGGRAPH 2015).* (Aug. 2015). DOI: 10.1145/2766997.

[17] Jonathan Korein and Norman Badler. "Temporal Antialiasing in Computer Generated Animation". In: *ACM SIGGRAPH Computer Graphics* 17 (July 1983), pp. 377–388. DOI: 10.1145/800059.801168.

[18] Henry C Lin and Andrew Burnes. "NVIDIA DLSS 3: AI-Powered Performance Multiplier Boosts Frame Rates By Up To 4X". In: (Sept. 2022). URL: https://www.nvidia.com/en-us/geforce/news/dlss3-ai-powered-neural-graphics-innovations/.

[19] Michael Oren and Shree K. Nayar. "Generalization of Lambert's Reflectance Model". In: *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (July 1994), pp. 239–246. DOI: 10.1145/192161.192213.

[20] Yaobin Ouyang et al. "ReSTIR GI: Path Resampling for Real-Time Path Tracing". In: *Computer Graphics Forum* (June 2021). DOI: 10.1111/cgf.14378.

[21] Hanspeter Pfister et al. "Surfels: Surface Elements as Rendering Primitives". In: (2000). DOI: 10.1145/344779.344936.

[22] R. James Purser, Manuel de Pondeca, and Sei-Young Park. "Construction of a Hilbert curve on the sphere with an isometric parametrization of area". In: (June 2009). DOI: 10.13140/RG.2.1.4556.9440.

[23] Colin Riley and Thomas Arcila. "FIDELITYFX SUPERRESOLUTION 2.0". In: (Mar. 2022). URL: https://gpuopen.com/gdc-presentations/2022/GDC_FidelityFX_Super_Resolution_2_0.pdf.

[24] Martin Roberts. "Evenly distributing points on a sphere". In: (Aug. 2018). URL: http://extremelearning.com.au/evenly-distributing-points-on-a-sphere/.

[25] Lili Sang. "Fast & Beautiful Surfel Rendering on a Smartphone". In: (Apr. 2017). URL: https://engineering.matterport.com/fast-beautiful-surfel-rendering-on-a-smartphone-c04f30437f68.

[26] Thomas Schops, Torsten Sattler, and Marc Pollefeys. "SurfelMeshing: Online SurfelBasedMesh Reconstruction". In: *IEEE Transactions on pattern analysis and machine intelligence* (Sept. 2018). DOI: 10.1109/TPAMI.2019.2947048.

[27] Kenneth E. Torrance and Ephraim M. Sparrow. "Theory for off-specular reflection from rough surfaces". In: *Journal of the Optical Society of America* (Sept. 1967). DOI: 10.1364/JOSA.57.001105.

[28] Eric Veach and Leonidas J. Guibas. "Optimally Combining Sampling Techniques for Monte Carlo Rendering". In: *ACM* (1995). DOI: 10.1145/218380.218498.

[29] Open Source Computer Vision. "Optical Flow". In: (). URL: https://docs.opencv.org/3.4/d4/dee/tutorial_optical_flow.html.

[30] Tim Weyrich, Hanspeter Pfister, and Markus H. Gross. "Rendering deformable surface reflectance fields". In: *IEEE Transactions on Visualization and Computer Graphics* 11 (2005), pp. 48–58. DOI: 10.1109/TVCG.2005.14.

List of Figures

List of Tables

Selbstständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Seitens des Verfassers bestehen keine Einwände die vorliegende Masterarbeit für die öffentliche Benutzung im Universitätsarchiv zur Verfügung zu stellen.

Ort, Abgabedatum	Unterschrift des Verfassers