Relighting for 2D Copy&Paste

Lukas Murmann

Supervisor: Jan Kautz

MSc Computer Graphics, Vision and Imaging

September 2013

This report is submitted as part requirement for the MSc Degree in Computer Graphics, Vision and Imaging at University College London. It is substantially the result of my own work except where explicitly indicated in the text.

Department of Computer Science

University College London

Abstract

A frequent task in image editing is to copy a single object from a source image into a target scene. Users matte the object by hand or with a smart selection tool to ensure that no background pixels are copied over. Often however, matting is not enough and the result just does not look right. One particular problem occurs when source and target image were captured in different lighting environments. In this case, the shading on the copied object is implausible, and the editing operation easily spotted by a human observer.

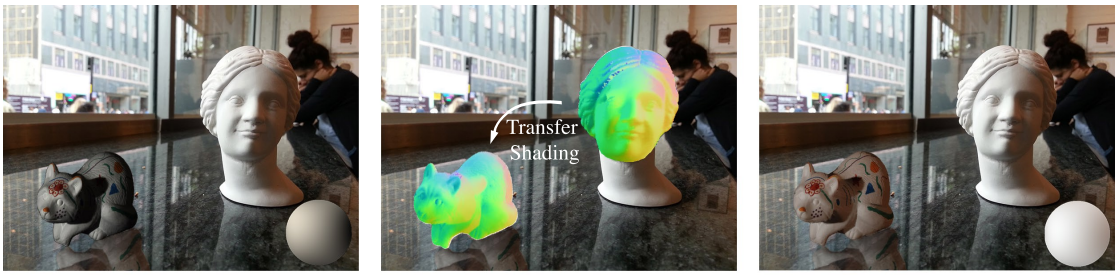
In this work, we estimate the lighting environment from only a single input image, and relight the copied object accordingly. As an intermediary step in relighting, we estimate both the object geometry and light environment. We base our method on a coarse initial geometry from which we detect lighting. In a second refinement step, we then detect fine surface detail.

We validate our technique in a number of relighting applications, where it achieves plausible results for a large class objects and light environments. We also report benchmark results of our geometry estimates that that show our algorithm on par with recent work in the field.

Contents

Chapter 1

Introduction



Not Relit | Geometry Estimate | Relit

Figure 1.1: A common image editing workflow: **Left:** We mask a source image and paste it into a target scene. However, the inserted object was captured under a different lighting environment than the target scene; it looks out of place. **Center:** Our algorithm estimates geometry and shading of the inserted object, and a second object in the target scene. We transfer shading to the inserted object. **Right:** The relit result. Raccoon image: [Grosse2009]

.

Manual photo relighting is challenging and time consuming, even for a skilled user. Figure 1.1 shows such an image-editing workflow. The raccoon has been captured under a strong directional light source. This results in a high contrast image with deep shadows. It has been matted—either manually or assisted by a segmentation algorithm such as GrabCut [Rother2004]—and was composited into a scene with softer illumination of a different color temperature. In this scene, the deep shadows on the inserted object look out of place. Our algorithm estimates shading and geometry of the inserted object, and transfers the scene's shading values to it. Figure 1.1 shows the relit scene.

The raccoon is now under softer light; the color temperature of the lighting matches the target scene.

Our algorithm also performs the inverse task: Given a possibly modified input image, it indicates whether the image has been tampered with or not. For this, the algorithm estimates the shading of two objects in the scene, in our example above the shading of marble figurine and the raccoon. In most legitimate photographs, we expect the two estimates to be similar. Significantly different shading on the other hand indicates image manipulation.

The third use case of our algorithm is related to the 2D compositing example we introduced first: Instead of inserting a 2D layer into a photograph, we can also render an existing 3D model into the scene. This is useful in augmented reality or visual effects applications. There are other techniques for this task (see section 2.1), but they require more input data and more user interaction than our algorithm. Our algorithm allows an unskilled user to render 3D models into photographs. We believe that as 3D acquisition tools like stereo camera systems [1] and range scanners [2] are becoming available to consumers, there will be increased interest in algorithms that combine 2D and 3D assets into new images.

We derive the required steps of our algorithm from the basic laws of light transport and reflection. They are modeled by the reflectance equation

$$L(\vec{x}, \vec{\omega}_o) = \int_{\Omega_{\mathcal{H}}} f_r(\vec{x}, \vec{\omega}_i, \vec{\omega}_o)\, L(\vec{x}, \vec{\omega}_i)\, \vec{n} \cdot \vec{\omega}_i \, \mathrm{d}\vec{\omega}_i \ , \tag{1.1}$$

a special case of the rendering equation [Kajiya1986] where we assume that the object does not emit light. See Section 4.1 for a detailed description of assumptions we make in our work. $L(\vec{x}, \vec{\omega}_o)$ is the reflected radiance captured by the camera. As a differential quantity, radiance is a function of both the position on the emitting surface $\vec{x}$ and the direction of the captured ray $\vec{\omega}_o$. In the same way, $L(\vec{x}, \vec{\omega}_i)$ is the light that arrives at the observed point $\vec{x}$. The surface normal at $\vec{x}$ is denoted by $\vec{n}$. The right hand side integrates the incoming radiance L_i, weighted with the surface reflectance (BRDF) f_r for all incoming light directions $\vec{\omega}_i$ over the hemisphere $\Omega_{\mathcal{H}}$.

[1] http://www.lg.com/uk/mobile-phones/lg-P920-optimus-3d
[2] http://www.fuel-3d.com/

Since we take only a single input photograph, the just described quantities—light, reflectance, and geometry— are unknown. Estimating these quantities, known as inverse rendering, is an underconstrained problem [Ramamoorthi2001a, Belhumeur1999] that will be the primary focus of our work. The report is structured as follows: We first (chapter 2) discuss how other researchers approach inverse rendering and relighting problems. We also look closer at computer vision methods that estimate light, material, or geometry properties. In chapter 3, we describe prerequisites for our algorithm in more detail. Section 4 is our core theoretical contribution: We derive an algorithm that estimates reflectance, light, and geometry, and synthesizes new images from those estimates. Results of this method are reported in 5. We report both qualitative results from applications such as relighting and forgery detection, and quantitative results about the accuracy of our geometry estimates. Finally, chapter 6 evaluates our results, summarizes our key contributions, and discusses possible avenues for future research.

Chapter 2

Related Work

Researchers have proposed a number of applications that allow users to estimate and change properties of their images. Some techniques render objects into existing photographs, others allow light or material parameters of a scene to be edited by the user. We discuss these applications in the first part of this chapter. In the second part, we then look closer at computer vision techniques that estimate lighting, reflectance, or geometry.

2.1 Render into Photographs

Rendering Synthetic Objects into Real Scenes This work by Paul Debevec [Debevec1998] introduced the idea of image-based lighting, a technique that uses HDR photographs as light sources in rendering. We will use the term *radiance map* for such images.

Once a radiance map is acquired, it is used to light a synthetic 3D object in a real photograph. This new object will in turn influence the scene: It will cast shadows, be reflected by glossy and specular surfaces, or might itself reflect onto nearby surfaces. Figure 2.1 shows such a scene.

In order to compute the mentioned global illumination effects, the geometry and material properties of the affected parts of the scene must be known. However, manually

Figure 2.1: **Left:** The original photograph. **Right:** Diffuse, glossy, and specular objects are rendered under real illumination. The objects are rendered with realistic interreflections and shadows. Image source: [Debevec1998]

adding this information for the entire scene is far too complex. Debevec hence makes a simplifying assumption: the inserted object affects only a subset of the scene. He calls this part the *local scene*. The local scene's geometry and material properties are then modeled by the user. With this added information, a global illumination renderer adds the indirect illumination to the photograph.

Both just mentioned contributions—image-based lighting and the notion of a local scene—are highly relevant for our work. We keep in mind though that we want to infer a radiance map directly from the image, not capture it as a separate HDR photograph. Also, want to minimize the amount of user interaction required to define the local scene.

Render into Legacy Photographs This work by Karsch et al. [Karsch2011] extends the ideas developed in the first "Render Synthetic Objects" paper [Debevec1998]. One notable drawback of this earlier work is that it requires access to the original scene: Without an image of a light probe in the same environment, we are not able to insert synthetic objects. This inhibits interesting applications. For example, we may want to insert objects into photographs first photographed without this manipulation in mind. Another application is in visual effects for movies: Many old movie productions have no or only very crude effects. If we can infer scene lighting and geometry from a video stream, we can add state-of-the-art effects to classic movies

Karsch et al. address this problem through a mix of computer vision techniques and user interaction. Their algorithm first estimates a coarse geometry of the scene which reduces the modeling effort required by the user. On the other hand, the user must specify light sources and their position explicitly. Figure 2.2 shows an example of

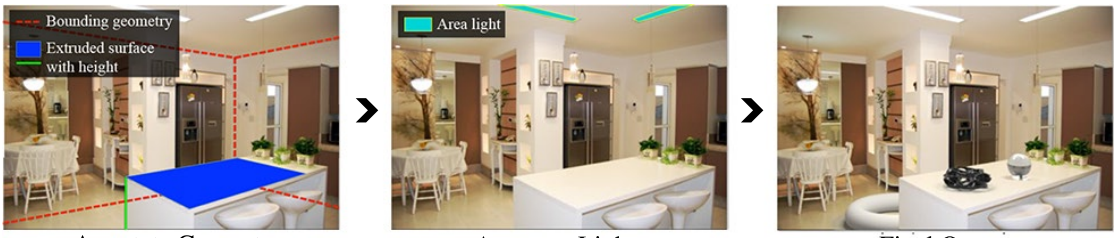


Figure 2.2: Rendering into Legacy Photographs. The algorithm first estimates basic scene geometry using computer vision techniques. Then, the user models all additional geometry by hand (left). The user also coarsely annotates light sources in the scene (center). On the right, we see the final render with inserted objects. Image source: [Karsch2011]

their technique.

Image-based Material Editing The goal of this work by Khan et al. [Khan2006] is not relighting. Instead, Khan et al. try to change the material of an object given only a single HDR photograph. They show examples of specular materials turned diffuse, diffuse materials turned metallic, and even make opaque objects appear transparent.

Khan et al. acknowledge that inverting the rendering equation is underconstrained. Hence, they do not look primarily for physically accurate solutions, but rather exploit weaknesses in human perception. For example, they directly interpret intensity values as depth, following a simple ”dark means deep” [Langer2000] model. Another simplification is in their lighting model. Like Debevec [Debevec1998], Khan et al. light their objects using radiance maps. However, these radiance maps are not acquired by photographing from light probes. Instead, the algorithm simply maps the image background to a sphere and uses it as a radiance map. It thus essentially uses the scene background as a light source.

The work by Khan et al. demonstrates that we can simplify the inverse rendering problem in a way that goes unnoticed by human perception. This result is of particular importance for us, since we can not expect to estimate every physical scene parameter correctly from only a single input image. Khan et al.’s results show that it can be feasible to render plausible synthesized images, even if intermediate parameters were not physically accurate.

Compositing Images Through Light Source Detection This work by Lopez-Moreno et al. from 2010 [LopezMoreno2010] and a recent extension from 2013 [LopezMoreno2013] has a problem setting similar to ours. Based on a single input image, they detect the scene lighting, and render a 3D model into the photograph. They also report results where the they estimate the geometry of a 2D object, and relight it to match the target scene.

In contrast to the techniques presented so far, Lopez-Moreno et al. do not use image-based lighting. Instead, they estimate an unknown number of point light sources from shading variations on the marked light probe. Indirect illumination is approximated with an additive ambient lighting term.

This method is conceptually simple. However, we argue that relighting an object with a discrete number of light sources cannot reach the same level of realism as image-based lighting does. In most real scenes, an object is not exclusively lit by direct light sources. Indirect illumination and interreflections from nearby objects play an important role that is inadequately expressed by a constant ambient term. Image-based lighting captures this indirect illumination and allows us to render global illumination effects even with very limited knowledge of the scene geometry.

2.2 Inverse Lighting

Inverse lighting algorithms estimate the light environment of a computer generated rendering or a real world photograph. Most inverse lighting algorithms require and input image, geometry, and reflectance data as input.

Light Representations Methods for inverse lighting differ in their representation of the light environment. One family of methods represents the incident light as 2D radiance maps and creates a light environment that can be used in image-based lighting. Other methods represent incoming light is as a discrete set of light sources; usually point lights or directional light sources. This representation is easy to render on most graphics pipelines, but a distinct disadvantage: Discrete light sources do not express global illumination effects such as interreflections between objects. Such affects are thus usually approximated by an *ambient* term.

Image-based lighting on the other hand does not describe where light originates, but rather how it arrives at a certain point. Consequently, both objects that emit light, and objects that merely reflect it, can be expressed in the same two dimensional function $L_i(\vec{\omega})$. For this reason, we will focus on image-based light representations in the remainder of this work. The original work by Paul Debevec [Debevec1998] serves as a good introduction to the topic of image-based lighting. More recent developments are collected in a book by Ward et al. [Ward2008].

Estimating Radiance Maps In 1997, Marschner and Greenberg [Marschner1997] estimated the continuous light environment by discretizing it using a set of piecewise constant basis functions. They thereby turn the reflectance equation (1.1) into a linear system. As long as the number of observed surface orientations is larger than the number of basis functions, the system is overconstrained and can be solved by least squares. However, Marschner and Greenberg noted that the linear system becomes increasingly ill-conditioned for diffuse materials.

Reflectance as Convolution In 2001, Ramamoorthi and Hanrahan expressed the reflection operation in a signal processing framework [Ramamoorthi2001a]. Equation 1.1 is seen as a convolution: The incoming light is the signal; it is convolved with the BRDF of the surface. A diffuse BRDF acts as a low-pass filter on the incoming radiance, while a perfectly specular BRDF acts as an all-pass filter that merely rotates the incoming radiance. With this insight, inverse rendering can be seen as the deconvolution of the rendering equation. It also explains why Marschner and Greenberg found it difficult to invert the lighting for diffuse materials: The low-pass characteristic of those materials removes high frequency variations in the incoming radiance; the deconvolution can hence only recover the smooth variations.

Until now, we assumed the material properties to be known. For the case where both material BRDF and incident radiance are unknown, Ramamoorthi and Hanrahan's work contains another important contribution: They derive for which combination of material and lighting the underlying factorization is well-posed, and propose a concrete algorithm to solve for both quantities.

Spherical Harmonics One question remains: Which set of functions is a suitable basis for such a signal processing framework? Convolution in cartesian coordinates is

efficiently computed using a Fourier basis, where it reduces to a dot product. However, BRDF and radiance are expressed in terms of a spherical frame (see Eq. 1.1), so the convolution theorem does not apply. The solution are spherical harmonics, a set of harmonic functions that relates to spherical coordinates similar like the Fourier basis related to the cartesian frame. Spherical harmonics are used in many fields of science, engineering, and were also used in computer graphics before. Still, the work of Ramamoorthi and Hanrahan, as well as independent work by Basri and Jacobs [Basri2003], sparked considerable interest in this basis. Subsequently, spherical harmonics were used in rendering [Ramamoorthi2001b, Sloan2002] and inverse lighting [Wu2011a, Barron2012a]. We provide more background on spherical harmonics in chapter 3.

2.3 Estimating Reflectance

Under controlled lighting conditions, a material's reflectance behavior can be estimated from captured photographs: Researchers in the field of appearance acquisition [Weyrich2008] capture the reflection of a light source for a large number of light positions and and derive physically accurate material properties from these images. The appearance acquisition literature contains important theoretical insights. However, in relighting applications, we do not have the same amount of control over the environment, and must deal with fewer input data. We thus focus this section on methods that estimate material properties from only a single input image. First, we discuss a technique where the scene's geometry is modeled by the user. Second, we consider intrinsic images: a family of algorithms that separate material reflectance from the interaction of light with geometry.

Material Acquisition From a Single Input Image with Known Geometry Such a system was proposed by Boivin and Gagalowicz [Boivin2001] in 2001. Figure 2.3 shows the input image with overlaid wireframe. Boivin and Gagalowicz's work is based on an optimization procedure that solves for the most likely reflectance function of each surface in the scene. A particularly interesting contribution is the optimization's error function: The authors make an assumption about material properties, render the image, and compute the pixel-wise distance between original photograph and rendered image.

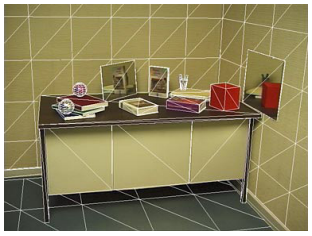


Figure 2.3: A photograph with user-specified geometry. Manual modeling of the scene geometry can take several hours. Image source [Boivin2001]

A downside of this approach is the need for accurate input geometry. Boivin and Gagalowicz report that modeling the geometry, camera position, and lights in the scene from Figure 2.3 took around six hours. We conclude that modeling the entire scene geometry is not an option in our scenario.

Intrinsic Images In 1978, Barrow and Tenenbaum [Barrow1978] introduced the idea of intrinsic images. In their general form, intrinsic images may capture a variety of scene properties: reflectance, specular highlights, or even surface normals. However, in this work we assume that intrinsic image algorithms extract only two separate images: A material reflectance image, and a *shading image* containing the lit geometry. See Figure 2.4 for an example.

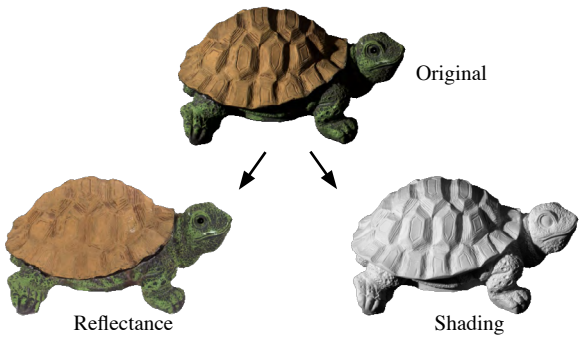


Figure 2.4: The original image is decomposed into two intrinsic images. Image source: MIT Intrinsic Image Dataset [Grosse2009]

A common idea among intrinsic image algorithms is to classify intensity gradients into either caused by reflectance or by shading. The Retinex algorithm [Land1971] implements this classification with a fixed threshold on gradient magnitude. It was later ex-

tended to color images [Kimmel2003]. In 2005, Tappen et al. [Tappen2005] applied machine learning to the gradient classification problem. They also used Markov Random Fields to smoothen the classification between neighboring pixels. We note though that the method has a training phase and therefore requires more than a single input image. In 2009, Grosse et al. published a ground truth data set for intrinsic images [Grosse2009]. The paper includes a survey and quantitative comparison of a number of intrinsic image algorithms including the three mentioned ones [Tappen2005, Land1971, Kimmel2003].

A very simple method for intrinsic images was proposed by Oh et al. [Oh2001] in 2001 and later used by Khan et al. for material editing [Khan2006]. THey use bilateral filtering [Tomasi1998] and split the input image into two layers: A layer of high-frequency intensity variations is assumed to contain the changes caused by reflectance. A base layer contains the low-frequency intensity variations attributed to shading.

Joint estimation of Reflectance, Lighting, and Geometry Separating material properties from shading is often a precursor to a subsequent shape from shading step where the geometry is recovered (see next section). Barrow and Malik [Barron2013] argue that the two problems are best solved jointly. They proposed SIRFS, a method that recovers shape, illumination, and material reflectance from a single RGB image [Barron2013].

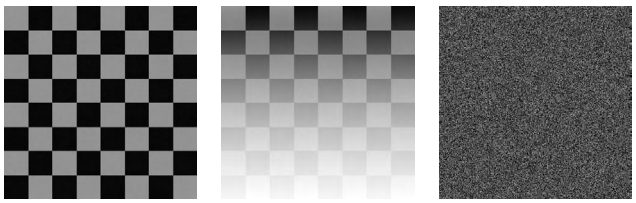


Figure 2.5: Prior knowledge restricts the reflectance intrinsic image to be both sparse and smooth. Surface textures are expected to most likely resemble the left image; it is thus favored in the underlying optimization. Image source: [Barron2012a]

The method of Barrow and Malik works with very little input data and provides all the results we need to relight diffuse objects. We thus take a closer look. Barrow and Malik acknowledge that jointly inferring reflectance, geometry, and lighting is severely underconstrained. They thus impose additional constraints: The reflectance

is constrained to be sparse and smooth (see Fig. 2.5). The geometry is constrained to be a closed surface, and is encouraged to have constant mean curvature. In practice, this makes smooth, spherical geometry more likely. Illumination is expressed using spherical harmonics up to order 2, which implicitly imposes a smoothness constraint on illumination [Ramamoorthi2001a]. We will use the reflectance retrieved by the SIRFS algorithm in later chapters. In chapter 5, we compare our geometry estimates to their results.

2.4 Geometry Estimation

We discussed methods that extract an intrinsic shading image from an RGB photograph. This shading image can now be used to infer geometry information in a technique called *shape from shading*. These shape from shading techniques are the first class of algorithms we discuss in this section. In the second part, we consider an alternative cue: shape from contour. Finally, we introduce techniques that use shape from shading on top of existing geometry: an approach we refer to as shading-based refinement.

Shape From Shading Shape from shading (SFS) has been studied in computer vision since the early 1970s [Horn1970]. The goal of SFS is to recover a height map from a single greyscale image. SFS algorithms usually work with a number of assumptions: Surfaces are assumed to be diffuse, the viewer is assumed to be distant, and the lighting is often assumed to be a known distant point light. A survey by Zhang et al. from 1999 [Zhang1999] gives a description and compares 6 algorithms. Another survey from 2008 [Durou2008] includes more recent work and also points to more general SFS algorithms that model perspective projection or specular materials. However, even with the strict assumptions of classical SFS, the problem is in general underconstrained. One of the reasons is the bas-relief ambiguity, which was pointed out in 1999 by Belhumeur et al. [Belhumeur1999].

The bas-relief ambiguity is not the only issue we face when we want to base our geometry estimation solely on shape from shading: most inverse lighting algorithms require the geometry to be known. This leads to a chicken-egg problem if SFS is in turn requires the scene lighting as input. In the following section, we will hence look into a

way of estimating geometry without knowing the lighting first.

Shape From Contour and Interactive Methods Shape from contour algorithms are found for example in interactive sketch-based modeling systems. Here, in contrast to conventional CAD modeling tools, users do not explicitly control the shape through vertices or control points. Instead, they sketch the object using a pen interface. The software then inflates the sketches into 3D representations.

One of the earliest of such systems is *Teddy*, proposed by Igarashi et al. in 1999 [Igarashi1999]. A recent work that was inspired by *Teddy* is Repoussé by Joshi and Carr [Joshi2008]. Repoussé describes a robust inflation method based on variational mesh processing [Botsch2004]. The beauty of such techniques is that we do not have to define a procedural algorithm to do the inflation: all we have to do is to set up a linear system; the actual computation is delegated to linear algebra software. For this reason, we chose a technique based on Repoussé as our shape from contour algorithm.

Shading-based Refinement Shape from shading algorithms can estimate high-frequent geometry details. Nehab et al. [Nehab2005] note that the quality of the normals computed with shape from shading even exceeds the quality of normals computed from laser scan or multi-view stereo results. Beeler at al. [Beeler2010] applied this shading-based refinement technique to performance capture. Here, a coarse geometry estimate is captured from multi-view stereo cameras. Since this geometry has low resolution and does not capture wrinkles and other pore-level detail important to their application, they recover such detail from shading variations.

In 2011, Wu et al. [Wu2011a] and 2012 Valgaerts et al. [Valgaerts2012] showed that shading-based refinement works with arbitrary, uncontrolled lighting (see Figure 2.6), and a single pair of consumer-grade cameras. Our work takes inspiration from their approach. Still, our problem differs in several ways: Our initial coarse geometry is much less detailed than the multi-view stereo techniques Wu and Valgaerts used. Furthermore, we have only access to a single input image, while performance capture applications usually draw from an entire sequence of frames.

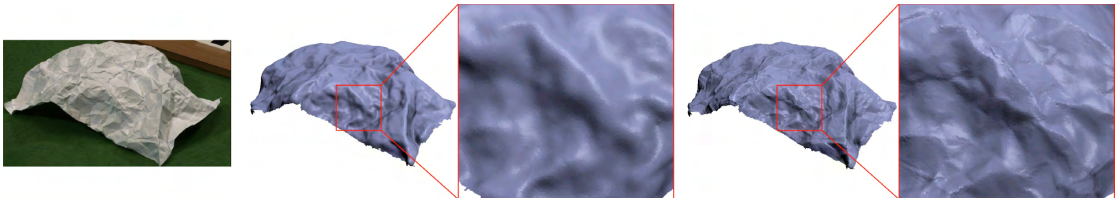


Figure 2.6: Wu et al.'s shading-based refinement approach. From left to right: The original photograph, a coarse initial shape from multi-view stereo, the shape with added high-frequency detail. Image source: [Wu2011a]

Chapter 3

Background

This chapter discusses genera mathematical tools and image processing concepts. While these topics are relevant to our later discussions, they are not specifically related to relighting. By discussing the beforehand, we keep the report self-contained, but do not have to draw attention from the main ideas in subsequent chapters.

3.1 Spherical Harmonics

This section gives an overview on the theory of spherical harmonics and summarizes the properties that are most relevant to our application. Our summary is based on discussions in classic textbooks [Jackson1998] and the computer graphics literature [Green2003, Jarosz2008].

Spherical harmonics are a orthogonal basis for a 2D field of directions over the sphere. Each spherical harmonics basis function encodes oscillation of a particular frequency. Hence, spherical harmonics are a harmonic transform—the spherical equivalent to the Fourier transform over 1D signals or 2D grids. Spherical harmonics basis functions Y_{lm} are assigned an order l and an orientation m. Parameter l is chosen to be a positive integer where higher orders denote higher oscillation frequencies; order 0 is a constant "DC term". Parameter m ranges from $-l$ to $+l$ so that each SH order contains two more basis functions than the next lower order. Figure 3.1 shows the resulting pyramid structure.

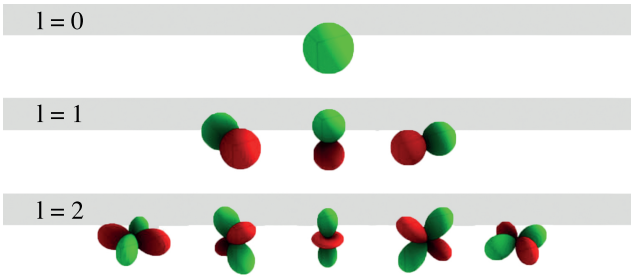


Figure 3.1: The first three orders of spherical harmonics. Positive values are shown in green, negative values in red. The radius of the shape illustrates the function's magnitude. Image source: [Green2003]

Project Function onto Spherical Harmonics Basis Spherical harmonics are a convenient representation for many rendering and relighting tasks. Often, we want to light a scene with natural illumination from the real world. Many such radiance maps are available online for research or for commercial applications [1]. However, these radiance maps are usually not stored in a SH representation, but as 2D images mapped to the spherical surface. This section introduces how to transform them into a SH representation.

We can project any function $f(\vec{\omega})$ onto the spherical harmonics basis by means of orthogonal projection

$$f_{lm} = \int_{\Omega} Y_{lm}(\vec{\omega})\ f(\vec{\omega}) \mathrm{d}\vec{\omega}. \tag{3.1}$$

with the inverse spherical harmonics transform

$$f(\vec{\omega}) = \sum_{l=0}^{\infty} \sum_{m=-l}^{l} Y_{lm}(\vec{\omega}) f_{lm} \tag{3.2}$$

Radiance maps are commonly stored as *equirectangular mappings*. An equirectangular mapping is a one-to-one mapping between the spherical surface and a rectangular image with W columns and H rows. Each row is assigned an elevation angle $\theta \in [0, \pi)$

[1] e.g. http://www.pauldebevec.com/Research/HDR/

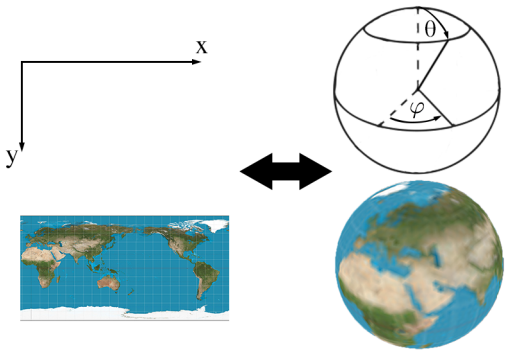


Figure 3.2: Example of an equirectangular mapping. Two vectors on the sphere span the same angle when mapped to the image plane. However, area measures in the image plane are increasingly distorted towards the poles.

and each column is assigned an azimuth angle $\varphi \in [0, 2\pi)$ such that

$$\varphi = 2\pi \frac{x}{W}$$
$$\theta = \pi \frac{y}{H}$$

and conversely

$$x = \frac{\varphi \cdot W}{2\pi}$$
$$y = \frac{\theta \cdot H}{\pi}.$$

Figure 3.2 illustrates this mapping.

In most relighting applications, we do not acquire the spherical harmonics coefficients by orthogonal projection as described in this section. The described method requires a large number of samples of $f(\vec{\omega})$ when computing the integral in Eq. 3.1. In relighting, we cannot easily integrate over the set of all directions Ω. For example, the visible front-facing normals might not cover all directions of the hemisphere; back-facing directions are missing entirely. Given such input data, we will estimate the spherical harmonics coefficients using linear regression. Section 4.5 details this step.

3.2 Gamma

Gamma is commonly explained as a relic from the days of CRT displays (e.g. [Slater2001, p111][Shirley2009, p62]). The electron ray inside such a monitor has a nonlinear relationship between input voltage and output intensity. The relationship is often modeled by a power law $Y \sim V_{\text{in}}^{\gamma}$ with a Gamma value of 2.4. If we want a linear relationship between values written to the framebuffer and the output luminance Y, we have to Gamma correct first: Say I is the computed image and I_{fb} is written to the framebuffer, the correction is $I_{\text{fb}} = I^{\frac{1}{\gamma}}$. This results in a linear relation between pixel values and output luminance of the CRT. When using modern LC screens that do not suffer from a nonlinear transfer function, either the operating system or electronics of the display simulate the old behavior and ensure compatibility with legacy software.

This view of Gamma correction is correct, but does not tell the whole story. The legacy of CRT displays is one reason for the nonlinear transform before we write to a framebuffer. The other reason lies in the human visual system, and in the nonlinear relation between luminance Y, and perceived lightness L^*. Lightness L^* is a perceptual measure defined by the CIE [CIE2004, Sec.8.2]. Poynton shows [Poynton2013] that the relationship between luminance and lightness can be approximated by the power-law function $L^* \sim L^{\frac{1}{2.4}}$. This allows for a second interpretation of the nonlinear display response: It corrects for the nonlinear behavior of the human visual system. As a result, values written to the framebuffer are roughly proportional to perceived lightness

$$I_{\text{fb}} \sim L^*.$$

Images loaded from the internet of from digital cameras are usually processed and have a Gamma precorrection of $I^{\frac{1}{\gamma}}$ applied to them. Unfortunately we often to not know which Gamma value exactly. Standardized color spaces like sRGB change that, but the image will still have gone through a number of color and contrast processing steps; we are unlikely to recover any physically meaningful luminance values. We thus allow the user to specify a reasonable Gamma correction value manually for preprocessed images. Often, a value of 2.4 is a good choice. However, as discussed above, there is also a point to be made for specifying a Gamma of $\gamma = 1.0$: With linear Gamma, we acknowledge the missing link to physical quantities, and try instead keep pixel differences

more closely to the intensity differences perceived by a human observer.

Chapter 4

Relighting from a Single Source Image

This chapter describes how we estimate shading and geometry of photographed objects. We sometimes express our ideas in mathematical formulas. Table 4.1 lists the symbols we use and their meaning.

Table 4.1: Symbols used throughout the text.

Symbol	Meaning
$a, b, c, \alpha, \beta, \gamma$	Scalars
$\mathbf{b}, \mathbf{x}$	Vectors
$\mathbf{A}, \mathbf{C}, \mathbf{D}$	Matrices
$f(\cdot)$	Scalar-valued function
$\vec{x}$	Point in 3D space
$\vec{n}$	Surface normal direction
$\vec{\omega}, \vec{\omega}_i, \vec{\omega}_o$	Light direction.
φ	Azimuthal angle $\in [0, 2\pi)$
θ	Elevation angle $\in [0, \pi]$
$Y_{lm}(\vec{\omega})$	Spherical harmonics basis function of order l and degree m
$\boldsymbol{Y}(\vec{\omega})$	Vector of spherical harmonics basis functions evaluated at $\vec{\omega}$
θ_{lm}	Coefficient in spherical harmonics basis
L	Radiance
$f_r(\cdot)$	BRDF
$\rho(\vec{x})$	Spatially varying reflectance factor in BRDF
ρ	Reflectance, BRDF for Lambertian material
$\mathcal{S}(\vec{n})$	Shading sphere

4.1 Assumptions

We already noted that t solving the inverse rendering problem from a single image is underconstrained in an underconstrained problem. Still, we can narrow the space of possible solutions by making assumptions about light transport, image formation, and by including prior knowledge about likely shapes.

Distant Light Sources By assuming distant light sources, we make sure the lighting is spatially constant. If we further ignore occlusions, the 5D light field $L(\vec{x}, \vec{\omega})$ is reduced to two dimensions: $L(\vec{\omega})$. This assumption is common among image-based lighting algorithms.

Distant Viewing Position Like most shape from shading algorithms, we assume the viewer to be distant. This is equivalent to assuming orthographic projection. The assumption is a common requirement for optimization-based techniques. It keeps the projection operation linear and which simplifies many formulas and techniques.

Diffuse to Glossy Materials We assume that our materials are diffuse or only slightly glossy, not specular. Estimating lighting from specular object is a different—often easier—problem than to estimate lighting from diffuse objects. However, our shading sphere representation (see section 4.1.2) is based on spherical harmonics, which are a very efficient and compact representation for low frequency functions, but inefficient for functions that contain wide frequency bands. Unfortunately, the shading sphere of a specular material might contain very high frequencies, which makes them difficult to handle in our framework.

Smooth, Sphere-like Objects We want to estimate the coarse geometry of our object based on its contour. We thus follow the assumption of contour-based techniques [Igarashi1999, Barron2012a, LopezMoreno2010] and assume smooth geometry with silhouette normals parallel to the image plane.

Separability of Spatially-varying BRDF Components We will call this the *intrinsic images assumption*. The general BRDF $f_r(\vec{x}, \vec{\omega}_o, \vec{\omega}_i)$ can be factored into a spatially varying reflectance image $\rho(\vec{x})$ and a spatially invariant modified BRDF $f'_r(\vec{\omega}_i, \vec{\omega}_o)$ such that $f_r(\vec{x}, \vec{\omega}_o, \vec{\omega}_i) = \rho(\vec{x}) f'_r(\vec{\omega}_i, \vec{\omega}_o)$.

4.1.1 Simplifying the Reflectance Equation

With the assumption of spatially constant incident lighting $L(\vec{x}, \vec{\omega}_i) = L(\vec{\omega}_i)$ the reflection equation 1.1 becomes

$$L(\vec{x}, \omega_o) = \int_{\Omega_{\mathcal{H}}} f_r(\vec{x}, \vec{\omega}_i, \vec{\omega}_o)\ L(\vec{\omega}_i)\ \ \vec{\omega}_i \cdot \vec{n}\ \mathrm{d}\vec{\omega}_i.$$

Unfortunately, there still is a spatial dependency in the BRDF $f_r(\cdot)$. We now use the intrinsic images assumption that states we can factor out the spatially-varying components of the BRDF. This turns the reflectance equation into

$$L(\vec{x}, \omega_o) = \rho(\vec{x}) \int_{\Omega_{\mathcal{H}}} f'_r(\vec{\omega}_i, \vec{\omega}_o)\ L(\vec{\omega}_i)\ \ \vec{\omega}_i \cdot \vec{n}\ \mathrm{d}\vec{\omega}_i.$$

Finally, we observe that ω_o is a constant under the orthographic projection assumption. We thus eliminate the constant and express the reflected radiance L_o solely with respect to the surface normal orientation at the observed point $\vec{x}$. Our simplified reflectance model is

$$\begin{aligned} L(\vec{x}, \vec{n}) &= \rho(\vec{x}) \int_{\Omega_{\mathcal{H}}} f'_r(\vec{\omega}_i)\ L(\vec{\omega}_i)\ \ \vec{\omega}_i \cdot \vec{n}\ \mathrm{d}\vec{\omega}_i \\ &= \rho(\vec{x}) \mathcal{S}(\vec{n}) \end{aligned} \tag{4.1}$$

with *shading sphere* $\mathcal{S}(\vec{n})$.

4.1.2 The Shading Sphere

Shading spheres represent the scene lighting convolved with the object's modified BRDF f'_r. When we relight, we directly transfer shading spheres, not scene illumination. We thereby avoid having to estimate scene illumination from diffuse materials, which is an ill-posed problem [Ramamoorthi2001a] that cannot be solved without additional prior information. Shading spheres are best thought of as *virtual light probes* with uniform reflectance; our best guess of how a real light probe in the scene would look like. Uniform reflectance does not mean that shading spheres have to be white: In scenes with colored incident lighting, the shading sphere will take on the color of the light source. Also note that shading spheres do not have to be Lambertian: The shading spheres we estimate take on the same *glossiness* as the material they are recovered

from. Once we have estimated a glossy shading sphere, we can synthesize less glossy and ultimately Lambertian versions by removing high frequencies.

There are two catches when it comes to glossy shading spheres: First, we can only transfer shading between objects of the same glossiness, or to objects that are less glossy than the source object. This can make it difficult to find an appropriate virtual light probe in a scene if we want to insert a glossy object. Second, we might find it difficult to get a reliable shading image from which we can estimate the shading sphere, since most intrinsic images algorithms assume materials to be Lambertian. We therefore, even though in theory not a restriction of our relighting method, focus on Lambertian materials in our work.

We will discuss the details of how to estimate shading spheres in section 4.5. Figure 4.1 shows three shading spheres of increasing glossiness, estimated under the same illumination.

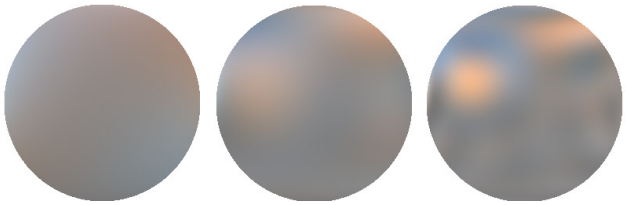


Figure 4.1: Shading spheres of different glossiness under colored illumination. The left sphere is Lambertian. The spheres in the center and on right are increasingly glossy. From a glossy sphere, we can always compute a less glossy one by low-pass filtering.

4.2 Relighting Algorithm Overview

Our algorithm breaks into a number of individual components. Figure 4.2 provides an overview over the general process, and shows the input and output data of each stage. In the following sections, we describe in detail how we inflate the initial shape estimate, how we estimate the shading sphere, and how we refine the shape estimate in a nonlinear optimization.

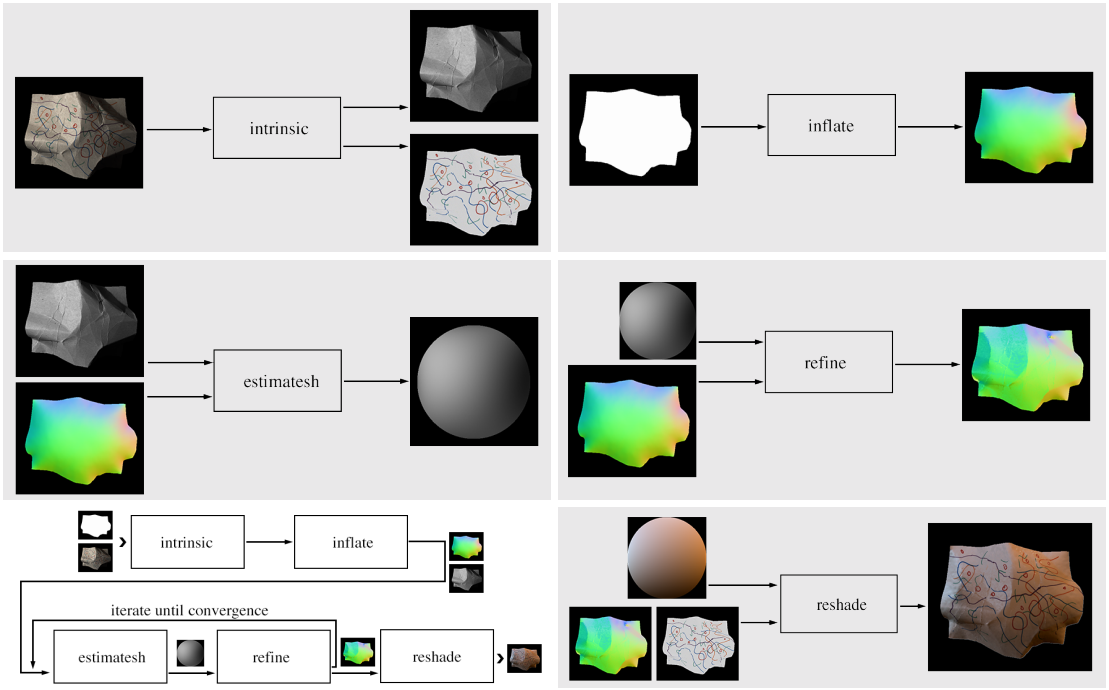


Figure 4.2: The individual steps of our algorithm. The flow diagram in the lower left provides an overview; the remaining five figures show the inputs and outputs of each phase. The first two steps are only run once: An intrinsic images algorithm separates shading and reflectance, and we estimate the geometry by inflating the shape's contour. From this basic geometry, we estimate the shading sphere using linear regression, and move on to refine the normals in a nonlinear optimization. These two steps are iterated until convergence. As a final step, we import the shading sphere from a second image and apply it to the refined geometry.

4.3 Separate Shading and Reflectance

Separation of the original input photograph into a shading and a reflectance layer is the first step in our algorithm. There is a wealth of related work (see section 2.3; we use one of the existing algorithms, rather than to create our own. We compared results of the Color-Retinex implementation provided with the MIT dataset, the "Intrinsic Images by Clustering" technique by Garces et al [Garces2012], and the result of the technique by Barron and Malik [Barron2013]. The latter provided the most consistent results across images from controlled and from uncontrolled environments.

The intrinsic images step contains an important modeling choice: Should we model the shading image—and consequently the scene lighting—as greyscale or RGB? A greyscale shading image implies the assumption of white light. This assumption is appropriate for white-balanced photographs that are dominated by a single type of light source. However, there are common scenes where the white-light assumption does not hold. For example, a photograph as shown in Figure 1.1 of an object by a window, when the interior is painted in an intense color. In such a scene, the part of the object facing the window will be illuminated by natural light, while the other side will take on the color of the room's interior. Both assumptions are valid, and our pipeline supports both greyscale and RGB shading images. For most scenes, however, we achieved better color reproduction when we modeled shading with three channels, and all the Figures in this report were rendered in the RGB shading model.

Looking back to our simplified reflectance model (see Eq. 4.1), we have now determined the reflectance image $\rho(\vec{x})$. The following sections will focus on recovering the shading sphere $\mathcal{S}(\vec{n})$ and the geometry, denoted by the set of surface normals $\{\vec{n}_i\}$.

4.4 Inflate Initial Geometry

The input to this step of the pipeline is a binary mask of a foreground object. As a first step, we turn this mask into a triangle mesh. We then inflate this mesh using a variational technique. The final result is computed by solving a sparse linear system.

Extract Outline Polygon Following the Repoussé paper [Joshi2008], we start by extracting the contour polygon of the masked object. This is a standard image processing problem, described for example in *Gonzalez&Woods 3rd ed.* [Gonzalez2006, Ch. 11.1.1]. Outline extraction results in an array of 2D-points. The contour is enumerated in clockwise order, and each point is given in absolute coordinates.

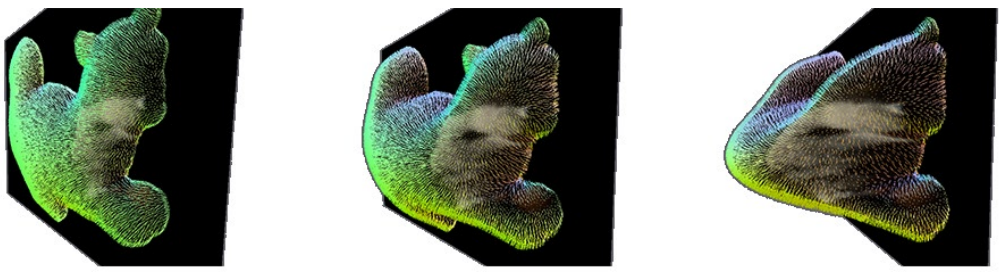
Find Triangulation After we converted the binary mask into a closed contour polygon, we turn it into a triangle mesh. An extensive body of research exists on how to generate high quality meshes [Bern1992]. We use Shewchuk's *Triangle* tool [Shewchuk1996] which provides us with a constrained Delaunay triangulation.

Triangle gives us fine-grained control over the generated mesh and lets us control the precision and computational complexity of later steps in the pipeline. First, we can specify the minimum triangle size. This ensures that we have enough vertices and hence enough data points for later processing. However, the complexity of most later processing steps scales linearly in the number of vertices. We therefore want to make sure that each vertex carries a relevant amount of unique information. Consequently, we chose a minimum triangle size such that there is roughly one vertex per input pixel. A second quality of *Triangle*'s triangulation is that we can specify a minimum angle size in the triangulation. Small angles indicate degenerate triangles which will lead to a poorly conditioned linear system in the inflation step. A minimum angle of 25 degrees worked well for us.

Laplacian Mesh Editing: Constant Mean Curvature We assume our geometry to be approximately spherical. We can encode this constraint by requiring constant mean curvature

$$H_i = \kappa$$

for all non-boundary vertices i and a user-defined constant κ. The curvature κ is the only user-specified parameter in our system. It has a very intuitive meaning: *The higher the curvature, the less flat the foreground object becomes*. Since a user can easily find a good κ interactively, we chose to make this parameter accessible to the user. See Figure 4.4 for an example.



Too Little Inflation | Correct Amount of Inflation | Too Much Inflation

Figure 4.3: The MIT *squirrel* test object with different amounts of inflation. The shape on the left is too flat and does not look realistic. The image in the center contains a good variety of normal directions and looks like we would expect a squirrel to look like. On the right, the mean curvature κ was chosen too high.

We now describe how these curvature constraints are implemented. The surface's mean curvature is related to its Laplace-Beltrami operator as follows [Botsch2010]:

$$\Delta_S \vec{x} = -2H\vec{n}$$

There are a number of ways to define a discrete approximation to the Laplace-Beltrami operator Δ_S on triangle meshes. We found a area-weighted operator

$$\Delta_S f(v_i) = \frac{1}{A_i} \frac{1}{|\mathcal{N}(v_i)|} \sum_{v_j \in \mathcal{N}(v_i)} f(v_i) - f(v_j)$$

where A_i is the neighborhood area of vertex i and $\mathcal{N}(v_i)$ is the set of neighboring vertices to yield good results. Normalizing by the neighborhood area makes the operator invariant to triangulation density. There are more robust definitions of the discrete Laplace-Beltrami operator, for example the cotan-weighted variant. These become necessary for non-uniform meshes with highly obtuse angles. Fortunately though, our meshes result from a constrained Delaunay triangulation which ensures uniform meshes; our use of the uniformly weighted Laplacian operator is valid.

We now solve for the x-coordinate (the distance from the image plane) of each vertex. The problem is takes the form of a Poisson equation

$$\begin{aligned} \Delta x &= \kappa, & v_i \in \Omega \setminus \partial\Omega \\ x &= 0, & v_i \in \partial\Omega \end{aligned}$$

with Dirichlet boundary condition on the foreground contour $\partial\Omega$. We write the problem as a system of linear equations. The system is sparse (usually 7 non-zero entries per row), which keeps the memory requirements of the system manageable, even for large meshes. Figure 4.4 explains how the system is set up. The following explains how we solve it.

Solving the Poisson Equation as a Sparse Linear System The problem is now of the form $\mathbf{Ax} = \mathbf{b}$ where A is sparse, square, and non-symmetric. Many efficient solving techniques however require the system to be symmetric and positive definite (SPD). We can make the problem SPD by solving $\mathbf{A}^T\mathbf{Ax} = \mathbf{A}^T\mathbf{b}$ instead. This proxy system can be solved efficiently, for example using Cholesky factorization.

As an alternative to solving the normal equations $\mathbf{A}^T\mathbf{Ax} = \mathbf{A}^T\mathbf{b}$, we can also look for an efficient solver that works with non-SPD systems. The iterative *Stabilized*

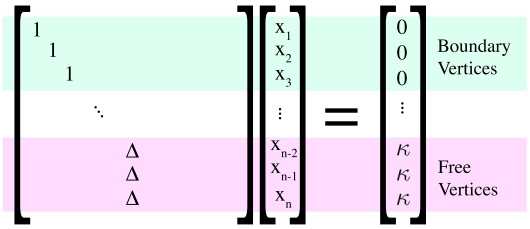


Figure 4.4: Sparse linear system for mesh inflation. Fixed boundary vertices are w.l.o.g. assigned consecutive indices starting at one. The resulting matrix has an identity submatrix in the upper left block and a zero matrix in the upper right. The lower rows model the free vertices: they are filled with weights of the Laplace-Beltrami operator and are assigned a target curvature κ.

Bi-Conjugate Gradient (BiCGSTAB) solver worked reliable on our problem. Botsch et al. provide more detail on the various sparse linear solvers [Botsch2005].

We compared the two mentioned techniques and settled on a direct solver based on Cholesky factorization. Depending on the size of the foreground object, it solves the problem in 3-5 seconds which is approximately 50% faster than the BiCGSTAB method. More importantly, factorization-based methods factor the system once, and can then solve it very efficiently for different right hand sides $\mathbf{b}$. Since the vector $\mathbf{b}$ captures the user-specified target curvature, Cholesky factorization is much better suited in an interactive environment where boundary conditions may change frequently.

4.5 Estimate Shading Sphere In Low-Order Spherical Harmonics

The previous step in our pipeline yielded a geometry estimate in form of a normal direction $\vec{n}_i$ at each vertex. From the input shading image, we can look up the target shading value s_i for each direction n_i. This lookup provides us with samples of the shading sphere $\mathcal{S}(\vec{n}_i) = s_i$. In this section, we will estimate the shading sphere using linear regression. More generally, we solve

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \sum_i (\mathcal{S}(\vec{n}_i, \boldsymbol{\theta}) - s_i)^2 \tag{4.2}$$

for the most likely lighting parameters $\hat{\boldsymbol{\theta}}$. We express the shading sphere $\mathcal{S}(\cdot)$ as a linear combination of spherical harmonics basis functions

$$\begin{aligned} \mathcal{S}(\vec{n}_i, \boldsymbol{\theta}) &= \sum_{l=0}^{L} \sum_{m=-l}^{l} Y_{lm}(\vec{n}_i)\theta_{lm} \\ &= \boldsymbol{Y}(\vec{n}_i)^T \boldsymbol{\theta} \end{aligned} \tag{4.3}$$

and substitute back into Eq. 4.2 which becomes

$$\begin{aligned} \hat{\boldsymbol{\theta}} &= \arg\min_{\boldsymbol{\theta}} \sum_i (\boldsymbol{Y}^T \boldsymbol{\theta} - s_i)^2 \\ &= \arg\min_{\boldsymbol{\theta}} \|\mathbf{A}\boldsymbol{\theta} - \mathbf{s}\|_2. \end{aligned} \tag{4.4}$$

This turns Eq. 4.2 into a linear least-squares problem with respect to the shading coefficients $\boldsymbol{\theta}$. The maximum order of spherical harmonics used to model the shading L is usually set to 2. This causes the estimated shading to be smooth, which is a reasonable assumption for diffuse and moderately glossy materials [Ramamoorthi2001a]. In the remainder of this section, we discuss how to best solve the resulting least-squares problem.

Estimate SH Coefficients Using Linear Least Squares In the previous equation, we introduced the *system matrix* $\mathbf{A}$. Assuming $L = 2$ and hence 9 SH basis functions, $\mathbf{A}$ is a $N \times 9$ matrix where N is the number of vertices. The problem of Eq.4.4 is solved by a theta that satisfies the normal equation

$$\mathbf{A}^T\mathbf{A}\boldsymbol{\theta} = \mathbf{A}^T\mathbf{s}.$$

Solving the Rank Deficient Linear System In cases where matrix $\boldsymbol{A}$ has full column rank, matrix $\boldsymbol{A}^T\boldsymbol{A}$ is positive definite. Unfortunately though, matrix $\mathbf{A}^T\boldsymbol{A}$ is often poorly conditioned. Some of its Eigenvalues become very small, which causes standard solvers based for example on Gaussian elimination to fail.

We achieved good results using a SVD-based solver. While other method like iterative BiCGSTAB were a bit faster, the SVD solver was more robust when we increased the maximum order of our SH basis functions. This is particularly useful for glossy materials, where the shading sphere may contain higher frequencies than covered by spherical harmonics of order 2.

Estimating shading at maximum order $L = 2$ is computationally not too complex, as $\mathbf{A}^T\mathbf{A}$ is a relatively small 9×9 matrix. In fact, the SH evaluations required to construct matrix $\mathbf{A}$ turned out to take more time than solving the resulting system.

Now that we have an estimate for $\boldsymbol{\theta}$ and the inflated geometry, we can go ahead and evaluate the shading equation $L_o(\vec{x}, \vec{n}) = \rho(\vec{x})\mathcal{S}(\vec{n}, \boldsymbol{\theta})$ for the first time (see Figure 4.5). The reflectance function ρ and the shading sphere $\mathcal{S}$ have plausible values, but the result does not look realistic yet: our geometry is still the inflated shape without any high-frequency detail. We will add this in the next step.

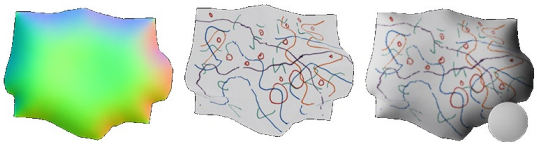


Figure 4.5: Intermediate result before the refinement step. Left we see the set of normals $\vec{n}$ from the shape inflation. In the center the reflectance function $\rho(\vec{x})$. In the right, we see the shading equation 4.1 evaluated for the estimated shading sphere $\mathcal{S}$.

4.6 Refine Normals in Nonlinear Optimization

The shading sphere $\mathcal{S}(\vec{n})$ we computed in the previous step is surprisingly accurate, even if the inflated geometry is only a very coarse estimate. We will now use the shading sphere to refine each normal $\vec{n}_i$ according to

$$\vec{n}_i = \arg\min_{\vec{n}_i} |\mathcal{S}(\vec{n}_i) - s_i|. \tag{4.5}$$

A Nonlinear Problem The two optimization problems in equation 4.2 and equation 4.5 are fundamentally different: While function $\mathcal{S}(\vec{n}) = \mathcal{S}(\vec{n}, \boldsymbol{\theta})$ is linear in the shading coefficients $\boldsymbol{\theta}$, it is nonlinear with respect to the normal direction $\vec{n}$. We thus cannot optimize using the linear least squares approach from the previous section. Also, the refinement problem has usually more than a single local minimizer: the outcome of common local optimization techniques will depend on our choice of initial value $\vec{n}_{i,0}$.

An Underconstrained Problem There is another key difference between problems 4.2 and 4.5. In the first, we used information from each vertex to determine a

single set of parameters $\boldsymbol{\theta}$. The problem was overconstrained. In the geometry refinement problem however, we consider each vertex individually. A single equation is used to infer a minimizer over a 2D domain. In total, we use N equations to estimate $2N$ unknowns. This is in general an underconstrained problem: we cannot expect the optimization procedure to converge.

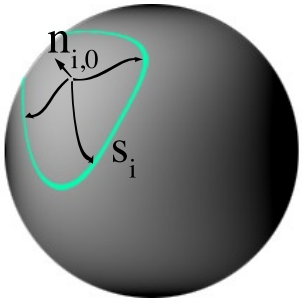


Figure 4.6: Without additional penalty term, the normal refinement problem is underconstrained. Any point along the marked isocontour is a viable minimizer $\hat{n}_i$.

Figure 4.6 visualizes the problem. Normal $\vec{n}_{i,0}$ of vertex i is the result of the inflation step; it becomes the initial value in the optimization. In the displayed case, the inflated normal does not explain the shading in the estimate s_i exactly, so the nonlinear optimization will move the estimate along the gradient of the shading sphere until it reaches a minimizing direction $\hat{n}_i$. Unfortunately, there are infinitely many such directions: they all fall on an isocontour as visualized in the figure. We can not tell for sure along which path $\vec{n}_i$ will move. Maybe even worse: The minimization might not converge. Instead, it will continue to move along the isocontour on it's search for an isolated minimizer

In order to constrain the refinement procedure further, we have to include prior knowledge in form of an additional penalty term. Two intuitive options are *smoothness* and *integrability* priors. A smoothness prior could for example be implemented in terms of a Markov Random Field: In addition to the cost function from Eq. 4.5, we would include *pairwise costs* that discourage a vertex from differing too much from neighboring vertices. In doing so however, we would have to make sure to still allow for edges in the refined normal field. A second constraint we might want to impose is integrability: We assume the modeled surface to be a height map, and restrict normal maps to

configurations that, when integrated, form a continuous surface.

Unfortunately, both mentioned priors—MRFs and integrability—are hard to implement as penalty terms in our nonlinear optimization. While these constraints might be a promising area for future research, we chose to implement a simpler prior that fits into our existing optimization code more naturally: We encourage normal estimates $\hat{n}_i$ to stay close to the initial direction $\vec{n}_{i,0}$. Looking at the example of Figure 4.6, this means that the optimization will converge to the point on the isocontour that spans the smallest angle to the initial direction. The modified optimization problem is now

$$\hat{n}_i = \arg\min_{\vec{n}_i} |\mathcal{S}(\vec{n}_i, \boldsymbol{\theta}) - s_i| + cos^{-1}(\ \vec{n}_i \cdot \vec{n}_{i,0}\ ), \tag{4.6}$$

where $cos^{-1}(\ \vec{n}_i \cdot \vec{n}_{i,0}\ )$ is the penalty term that increases as we move away from the starting direction $\vec{n}_{i,0}$. This penalty term works reliably and adds only little computational cost. However, it biases the refined solution towards the inflated geometry, and is thus a more arbitrary choice than priors based on smoothness or integrability.

Solving the Problem We solve problem 4.6 using the Levenberg-Marquardt algorithm with L_1 norm as implemented in the Ceres Solver package [Agrawala2012]. The book *Numerical Optimization* by Nocedal and Wright [Nocedal2006] provides a good introduction to nonlinear optimization and was the basis for our choice of optimization technique. We optimize each vertex normal sequentially. Overall, this nonlinear optimization is the computationally most expensive operation in our pipeline. Refining a mesh of $18,000$ vertices takes roughly 22 seconds on our development laptop.

From this refined geometry, we can now estimate a more accurate shading. With the improved shading, we refine the normals a second time. Experience shows that the algorithm converges after a small number of such iterations. In total, we can expect the algorithm to take about 60 seconds for joint shading and geometry estimation.

Chapter 5

Results

Based on the method described in section 4, we developed a scriptable set of command line tools and an interactive GUI application. We begin this results section by introducing these two applications. We then report results of two variations of our relighting algorithm: First, rendering of 3D meshes into existing photographs. Second, we describe how we turn our relighting pipeline into a *relighting detector*; an algorithm that indicates whether a photograph is legitimate, or manipulated. Finally, we report benchmark results of our geometry estimation on objects from the MIT data set.

5.1 Image Relighting as Automated Workflow

In section 4, we described how our algorithm breaks down in a number of sequential steps. We implemented each step as an independent command line tool. The *estimatesh* tool for shading estimation tool is one example. It is called from the command line with the following usage

```
$ ./estimatesh [-L<n>] -n <normals.png> -s <shading.png> -o <sh_coeffs.mat>.
```

This tool takes a shading image and a normal map as input. It reads the normal map from an RGB file, where the RGB channels map to XYZ coordinates. It runs the algorithm described in section 4.5 and returns a matrix of shading coefficients. This matrix is then read again by tools later in the pipeline. The *reshade* tool performs the final step: it combines the estimated geometry, reflectance, and the shading sphere of the target scene

into a new image. For example, Figure 5.1 shows the *MIT paper2*[Grosse2009] object lit by the *Pisa*[1] environment. Figure 5.5 shows more objects from the MIT dataset in three different lighting environments.

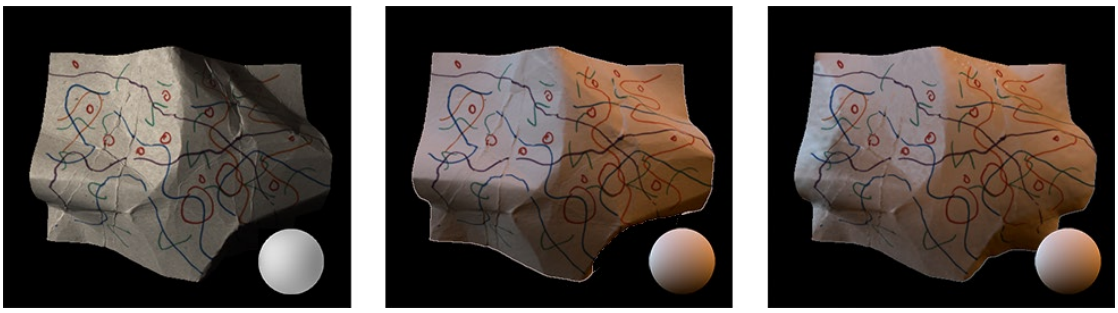


Figure 5.1: Relighting an image from the MIT data set. **Left:** The original image with estimated shading sphere. **Center:** Ground truth geometry and shading sphere of the Pisa environment. The artifact in the lower right is due to missing data in the geometry ground truth. **Right:** Our geometry estimate relit in the Pisa environment.

5.2 Interactive GUI Tool: Relighting with Visual Feedback

The tools described so far are a useful for automating the relighting process. In some cases however, we want to control the algorithm with more direct visual feedback. We have written a GUI application for this interactive use case.

The workspace for our interactive tool is shown in Figure 5.2. On the left is the main preview window (a). In the depicted scene, the foreground object has been replaced by a 3D mesh. We inflate this mesh by clicking the inflate button. Clicking again will inflate the mesh a bit more. We repeat this until the result looks satisfying. We then choose how many orders of spherical harmonics to use in our shading estimation. As discussed in section 3.1, allowing spherical harmonics basis functions up to order 2 is a good choice for Lambertian materials. We then press solve which solves for the spherical harmonics coefficients using linear regression. The snapshot shown in figure 5.2 was taken after this step.

[1] http://gl.ict.usc.edu/Data/HighResProbes/

Now that we have an initial estimate on both shading and geometry, we can preview the results in the additional frames right of the primary window. In frame (b), we see a rendering of the shading sphere; in frame (c), the entire sphere surface as an equirectangular mapping. In frame (d), the tool renders the current estimate of the normal map.

Just like when using the command line interface, we refine the normals and iterate between refinement and shading estimation. Whenever we are satisfied with a result, we press save, which will store the current shading and geometry to disk.

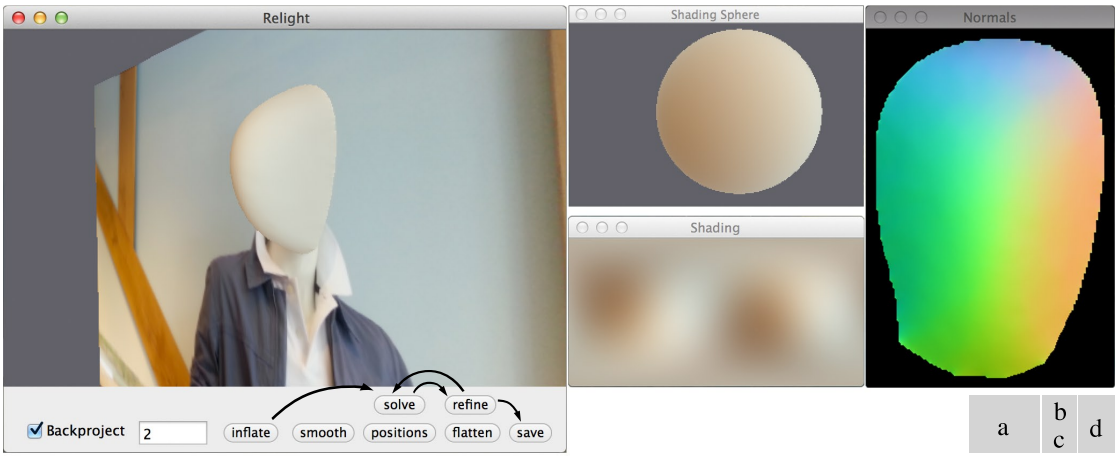


Figure 5.2: The GUI tool for interactive geometry estimation and relighting.

5.3 Rendering Synthetic Objects Into Real Scenes

Rendering synthetic objects is a variation of the relighting application we have seen earlier. Here, we do not estimate 3D geometry, but render an existing 3D model into a photograph. This has done before, but existing work either requires access to the original scene the photo was taken in [Debevec1998], or it requires the user to diligently model light sources and occluders in the scene [Karsch2011]. Our algorithm estimates the scene's lighting environment without direct user interaction.

We described in section 3.1 how shading environments are encoded using only a small number of spherical harmonics basis functions. The corresponding shading coef-

ficients are the only unknowns in this problem; geometry and reflectance information are defined by the 3D model. This narrows the space of possible solutions significantly and makes rendering 3D meshes a simpler problem than relighting 2D images. Figure 5.3 shows models from the Stanford model Repository [Curless1996] inserted into our test images.

We could further increase the level of realism by also rendering shadows cast by the inserted object. We omitted this step for two reasons: First, in order to render shadows accurately, we need access to the scene geometry, or have to ask the user to model relevant geometry by hand. Second, we do not know the light configuration that needs to be known for existing shadow algorithms. All we know is the shading sphere, which is a low-pass filtered version of the actual lighting. We believe that both problems can be overcome though: Instead of asking the user to model scene geometry, we can assume that the model stands on a flat surface. In many scenes, this assumption is valid, for example the scenes shown in Figure 5.3. We see shadow rendering from shading spheres as an interesting problem for future research.

5.4 Benchmark Results

In the previous section, we discussed our relighting workflow and showed how we relight the *paper2* test object. In this section, we analyze the quality of our estimated geometry and compare it to geometry estimates produced by the SIRFS [Barron2013] algorithm.

The original MIT dataset as published by Grosse et al. [Grosse2009] did not contain ground truth geometry data, only photographs taken under different illumination conditions. Barron and Malik addressed the problem of missing geometry in their SIRFS publication. They recovered surface normals using photometric stereo and made the resulting normal maps available for download. The normal maps are shown in the ground truth collumn of Figure 5.4.

We must choose an intrinsic images algorithm for our technique. In order to focus on the geometry estimation of both algorithms, we use the shading and reflectance images created by SIRFS as input to our algorithm. Also, we need to define an appropriate

Figure 5.3: Rendering a 3D meshes into photographs. We used the marble figurines as light probes and inserted models from the Stanford model repository.

error metric for our geometry estimate. We chose to compare the *mean absolute error* of normal angles between estimate and ground truth. Given two normals $\vec{n}_{\text{gt}}$ and $\vec{n}_{\text{est}}$ the formula for the MAE is

$$\text{N-MAE} = \frac{1}{N}\sum_{i=1}^{N} cos^{-1}(\ \vec{n}_{\text{gt}} \cdot \vec{n}_{\text{est}}\ ).$$

In Figure 5.4 we show the geometry estimates and N-MAEs of both SIRFS and our technique on 6 test objects from the MIT dataset. The two algorithms achieve similar benchmark results. Our technique performs better on 4 out of 6 tests, the *raccoon*, *paper2*, *teabag1*, and *squirrel* cases. SIRFS achieves a smaller error on the *turtle* and *sun* objects. In four of the tests, the difference is rather small. Results differ the most on the *teabag1* and *sun* objects. Our technique requires one user-defined parameter (the amount of inflation) while SIRFS includes prior information that was learned from other objects of the MIT dataset.

5.5 Tampering Detection

We previously estimated the light environment from one object and used it to light an inserted object. A variation of this technique is to estimate the light environment of two objects in the same image and compare the two environments [Johnson2007]. If the two light environments are substantially different, this indicates a possible image manipulation.

While incompatible lighting environments can be due to image manipulation, there are also legitimate reasons for them being different. For example, one of the objects might me lit, the other in shadow, or simply due to diffuse interreflections in the scene. In any case, we need a suitable error metric between two lighting estimates. One solution is to compute the distance in the space of spherical harmonic coefficients. However, this is unlikely to express the perceived distance between the two environments accurately. For example, two estimated light environments are allowed to differ on the side facing away from the camera: The difference will show in the coefficients, but can not be seen on any rendered surface. We therefore define the error metric over the rendered sphere. We render a white sphere under both illuminations, and take a per-pixel

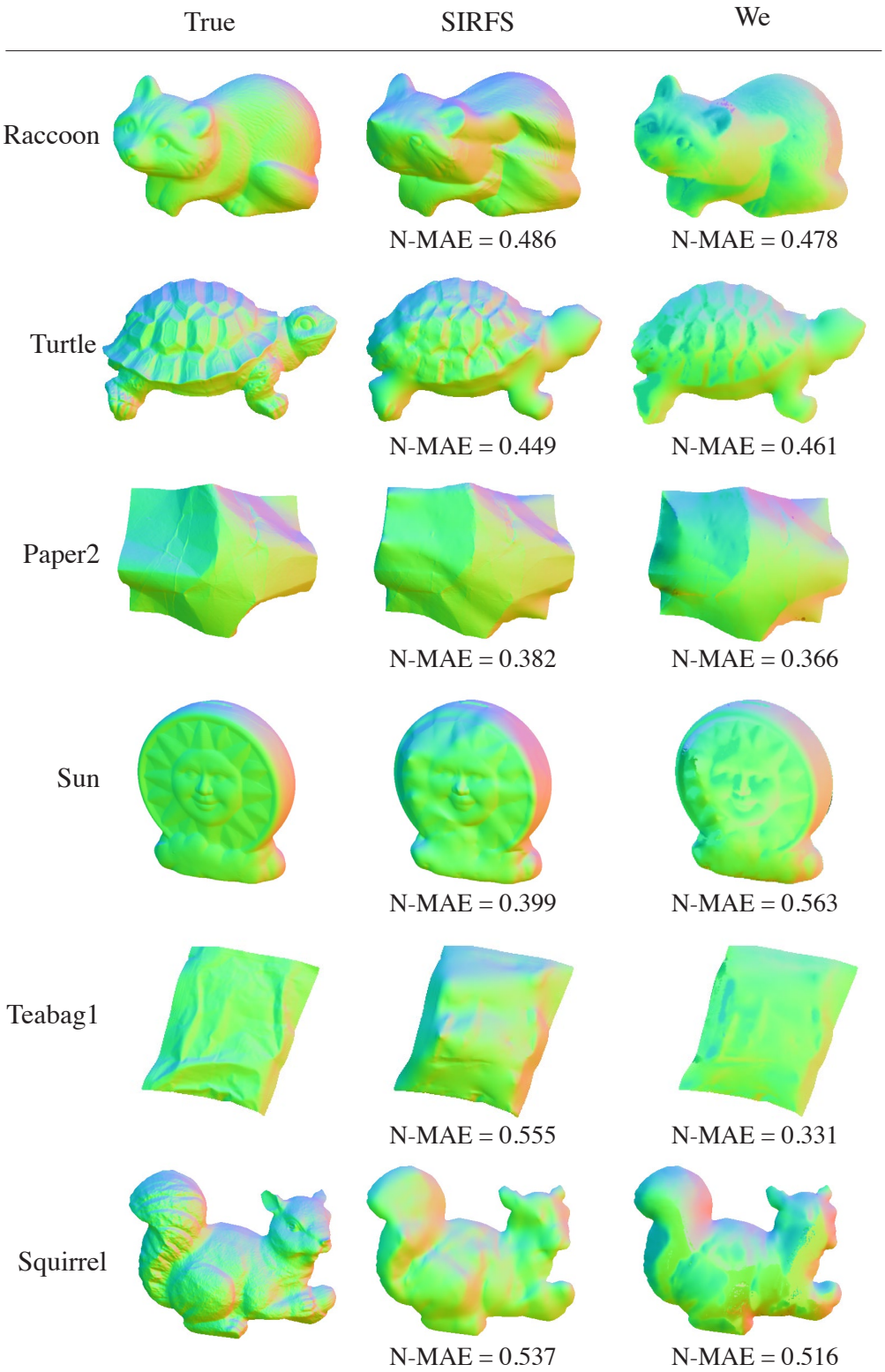


Figure 5.4: Geometry estimation results. We compare the mean absolute error of normal angles. Our method performs on par with the SIRFS [Barron2013] technique on objects from the MIT dataset.

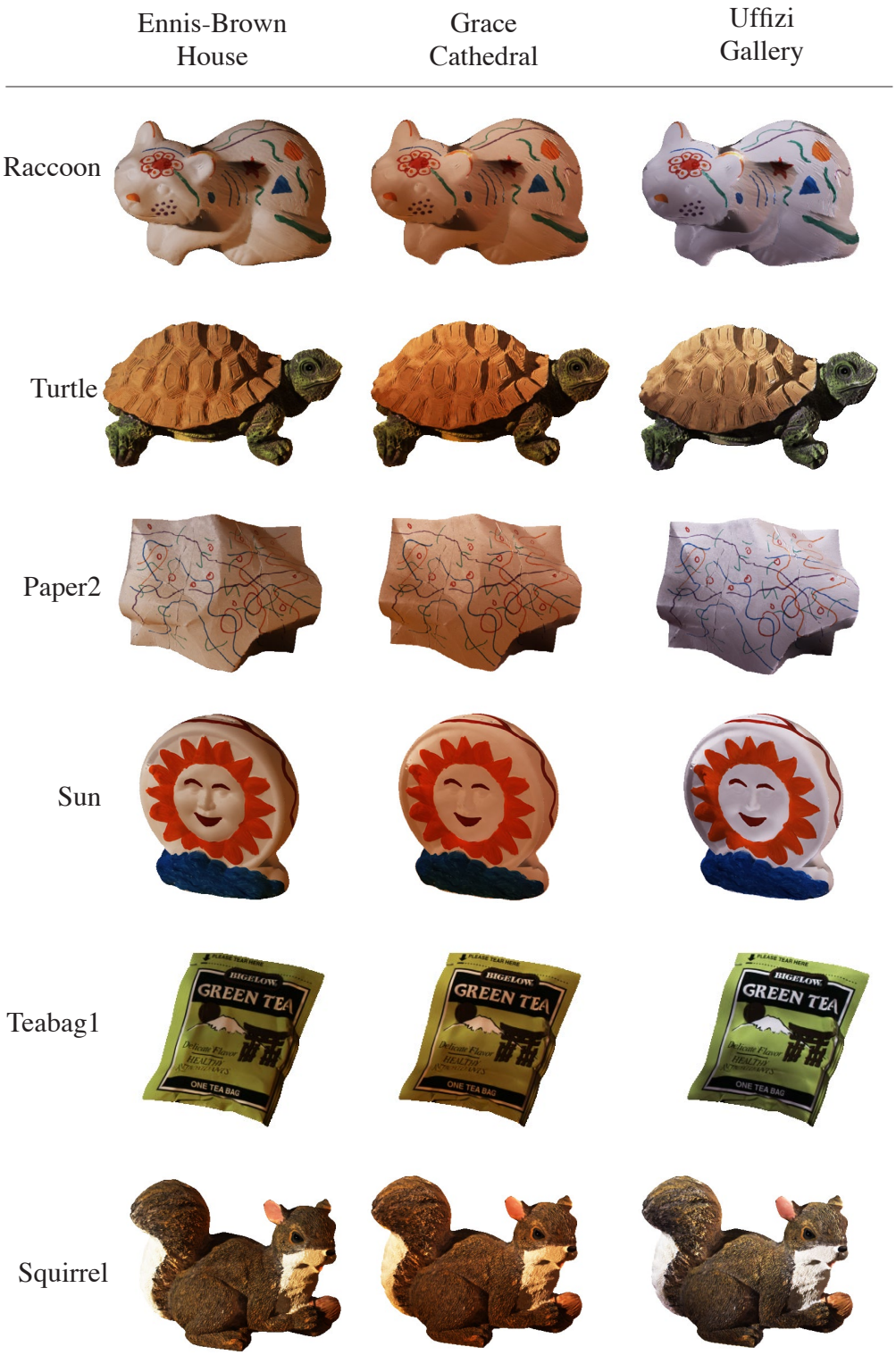


Figure 5.5: Relighting the MIT dataset. We relit objects from the MIT dataset in 3 different lighting environments. The geometry estimate is shown in Figure 5.4 in the right collumn.

distance.

Simply computing the mean squared error is not the best choice: The intrinsic images step is an underconstrained problem and the shading image is only defined up to a scalar multiple. We consequently compare the *scale-invariant* mean squared error between the two rendered shading spheres. Figure 5.6 shows the spheres and lists the error between the two.

In the first image, both objects belong to the same scene. In the second image, one object has been inserted. The error in the modified image is 10 times larger than in the legitimate one. It is easy for the algorithm to detect the manipulation.

Figure 5.6: Shading estimation for forgery detection. **Left:** This image is not tampered with. Both objects are under very soft illumination. **Right:** Here the right figurine was inserted. The scene was shot under direct sunlight from the direction of the camera. The right figurine was captured from the same direction, but without direct sunlight.

There is an interesting distinction between the full relighting workflow and tampering detection: In relighting, the foreground mask needs to cover the entire object that is to be relit. This works well as long as the object is convex and well approximated by a spherical shape. Unfortunately, the marble figurines shown in Figure 5.6 are not: they have concave regions and show significant self-shadowing. These are two properties that break the assumptions stated in section 4.1; they will lead to poor shading estimates. In shading estimation, we thus only include a convex subregion in the foreground mask. The masks we used are indicated in the bottom of Figure 5.6.

5.6 Discussion

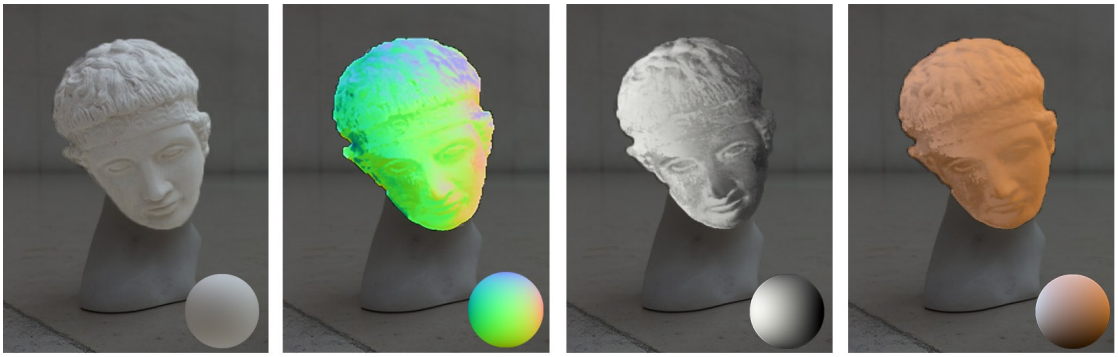
Relighting images from only a single input image is an underconstrained problem. As others have noted before [Grosse2009, Ramamoorthi2001a, Belhumeur1999], the subproblems of Intrinsic Images, inverse lighting, and shape from shading are all ill-posed. Still, by making the right assumptions, we can compute estimates that—while not physically accurate— lead to a perceptually plausible results. In the following, we will discuss the cases where our algorithm succeeds, and point out challenges and ambiguities it faces.

Quality of Estimated Shading Whenever the inflated shape approximates the true geometry well enough, our estimated shading sphere is a good model of the scene lighting. This holds for predominantly convex shapes, for example the *Turtle* test case from section 5.4. The turtle is a convex, sphere-like shape; concavities at the shell do not bias the lighting estimate too severely.

The marble figurines on the other hand are examples where concavities are not only found as surface detail, but span a significant portion of the shape. The concave neck region of these objects will cause the shading estimate to be useless for subsequent tasks. In the tampering application, we avoided this problem by creating a new mask that covered only the face of each figurine (see also Figure 5.7 where we used the same mask).

Quality of Estimated Geometry In some applications, estimating the shading sphere is not enough; we want to know the object's normals as well. Two of these applications are relighting and geometry estimation.

The normal maps estimated in section 5.4 are on par with other state-of-the-art algorithms. Still, some artifacts remain: For example, any self-shadowing effects—like the shadow cast by the raccoon's ear—lead to discontinuities in the normal field. Also, the refined geometry is biased towards the initially inflated shape; a result from the nonlinear optimization where we use the initial normal both as a starting direction, and where we added a penalty on the distance from the starting point. We see this bias in Figure 5.7 at the left of the *paper2* case and at the overhanging edge of the turtle's shell.



a b c d

Figure 5.7: The same geometry lit in 3 different environments. **a:** The original image with soft illumination from the right. **b:** The estimated geometry. **c:** The face relit under high-contrast illumination from the left. **d:** The same geometry lit in the *Pisa* environment. Note how in image d, the crease along the figurine's face—an artifact—is clearly visible, while it goes mostly unnoticed in image c.

In the relighting workflow from section 5.1, we applied a new lighting environment to the *paper2* object. Even though some of the normals are not estimated correctly, the rendered image in Figure 5.1 looks plausible. A similar case is shown in Figure 5.7. Here, we again estimate the normals of the marble figurine's head. The estimated normals are in general accurate, but there is a discontinuity along the left eye of the figurine. In one lighting environment, the discontinuity is barely noticeable. In a second environment however, it manifests as a distracting crease in the rendered output.

We can explain the discontinuities in the normal field by again considering Figure 4.6 from the refinement step: Usually, two neighboring vertex normals $\vec{n}_i$ will also converge to neighboring minimizers $\hat{n}_i$ on the isocontour where $\mathcal{S}(\hat{n}_i) = s_i$. However, if the two normals normals lie close to a stationary point of $\mathcal{S}$, they might converge to different segments of the isocontour. This causes a normal discontinuity that will become visible once we apply a new shading sphere.

We noted in section 4.6, that there are more sophisticated ways to include prior knowledge in the refinement stap. Integrability and smoothness are two examples. The analysis in this chapter reinforce our impression that these priors will be a rewarding area for future research.

Chapter 6

Conclusion

In this report, we discussed the problem of relighting objects given only a single input image. We motivated our work by describing a number of applications such as 2D copy& paste or tampering detection.

In chapter 2, we surveyed related work in the fields of inverse lighting, reflectance estimation, and shape from shading. We also reviewed applications with similar goals and approaches as our work. We went on (chapter 3) to provide background information on spherical harmonics, and discussed the implications of gamma correction on image processing algorithms.

Chapter 4 is our core theoretical contribution. Starting with a list of our assumptions, we simplified the general reflectance equation (Eq. 1.1) and arrived at the shading equation (Eq. 4.1). We then introduced the shading sphere, a key concept in our work, and moved on to describe our algorithm: First, we described how we inflate an initial shape estimate using a variational mesh editing technique. Next, we formulated shading estimation as a linear regression problem over spherical harmonic basis functions. From this initial shape and lighting estimate, we derived how to refine the geometry in a second, nonlinear optimization.

We applied our relighting method to a number of applications and report results in chapter 4. First, we described our command line tool chain and relighting results (Fig. 5.1 and Fig. 5.7). We also discussed a graphical tool for relighting that gives direct visual feedback. We introduced a second application, rendering 3D objects into

photographs, with models from the Stanford mesh repository. Next, we compared our geometry estimates with results of a state-of-the art algorithm [Barron2013]. The quantitative comparison shows that the estimates of both techniques have very similar accuracy. We believe that our method provides a viable alternative, in particular thanks to its modular structure: Research into any of the components will improve our system as a whole. We plan to do so in our future work, where we will first focus on more elaborate prior shape models for inflation and normal refinement.

Bibliography

[Agrawala2012] Sameer Agarwal and Keir Mierle. *Ceres Solver: Tutorial & Reference*. Google Inc. 33

[Barron2012a] Jonathan T. Barron. Shape, albedo, and illumination from a single image of an unknown object. In *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, CVPR '12, pages 334–341, Washington, DC, USA, 2012. IEEE Computer Society. 10, 12, 22

[Barron2013] Jonathan Barron and Jitendra Malik. Shape, illumination, and reflectance from shading. Technical Report UCB/EECS-2013-117, EECS, UC Berkeley, May 2013. 12, 25, 37, 40, 46

[Barrow1978] H. G. Barrow and J. M. Tenenbaum. *Recovering Intrinsic Scene Characteristics from Images*. Academic Press, 1978. 11

[Basri2003] Ronen Basri and David W. Jacobs. Lambertian reflectance and linear subspaces. *IEEE Trans. Pattern Anal. Mach. Intell.*, 25(2):218–233, February 2003. 10

[Beeler2010] Thabo Beeler, Bernd Bickel, Paul Beardsley, Bob Sumner, and Markus Gross. High-quality single-shot capture of facial geometry. In *ACM SIGGRAPH 2010 papers*, SIGGRAPH '10, pages 40:1–40:9, New York, NY, USA, 2010. ACM. 14

[Belhumeur1999] Peter N. Belhumeur, David J. Kriegman, and Alan L. Yuille. The bas-relief ambiguity. *Int. J. Comput. Vision*, 35(1):33–44, November 1999. 4, 13, 43

[Bern1992] Marshall Bern and David Eppstein. Mesh generation and optimal triangulation, 1992. 26

[Blinn1977] James F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, July 1977. 74

[Blinn1996] Jim Blinn. *Jim Blinn's corner: a trip down the graphics pipeline*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996. 64

[Boivin2001] Samuel Boivin and Andre Gagalowicz. Image-based rendering of diffuse, specular and glossy surfaces from a single image. In *Proceedings of the 28th annual*

conference on Computer graphics and interactive techniques, SIGGRAPH '01, pages 107–116, New York, NY, USA, 2001. ACM. 10, 11

[Botsch2002] M. Botsch, S. Steinberg, S. Bischoff, and L. Kobbelt. Openmesh - a generic and efficient polygon mesh data structure, 2002. 52

[Botsch2004] Mario Botsch and Leif Kobbelt. An intuitive framework for real-time freeform modeling. *ACM Trans. Graph.*, 23(3):630–634, August 2004. 14, 59

[Botsch2005] Mario Botsch, David Bommes, and Leif Kobbelt. Efficient linear system solvers for mesh processing. In Ralph Martin, Helmut Bez, and Malcolm Sabin, editors, *Mathematics of Surfaces XI*, volume 3604 of *Lecture Notes in Computer Science*, pages 62–83. Springer Berlin Heidelberg, 2005. 29

[Botsch2010] Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Levy. *Polygon Mesh Processing*. AK Peters, 2010. 28

[Bradski2008] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000. 52

[CIE2004] Commission Internationale de L'Eclairage. *15:2004, Colorimetry*. 3rd edition, 2004. 19

[Curless1996] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pages 303–312, New York, NY, USA, 1996. ACM. 37

[Debevec1998] Paul Debevec. Rendering synthetic objects into real scenes: bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 189–198, New York, NY, USA, 1998. ACM. 5, 6, 7, 9, 36

[Durou2008] Jean-Denis Durou, Maurizio Falcone, and Manuela Sagona. Numerical methods for shape-from-shading: A new survey with benchmarks. *Comput. Vis. Image Underst.*, 109(1):22–43, January 2008. 13

[Garces2012] Elena Garces, Adolfo Munoz, Jorge Lopez-Moreno, and Diego Gutierrez. Intrinsic images by clustering. *Comp. Graph. Forum*, 31(4):1415–1424, June 2012. 25

[Gonzalez2006] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. 26, 65

[Green2003] Robin Green. Spherical Harmonic Lighting: The Gritty Details. 2003. 16, 17

[Grosse2009] Roger Grosse, Micah K. Johnson, Edward H. Adelson, and William T. Freeman. Ground-truth dataset and baseline evaluations for intrinsic image algorithms. In *International Conference on Computer Vision*, pages 2335–2342, 2009. 2, 11, 12, 35, 37, 43

[Guennebaud2010] Gaël Guennebaud and Benoît Jacob. Eigen v3. http://eigen.tuxfamily.org, 2010. 52

[Horn1970] B. K.P. Horn. Shape from shading: A method for obtaining the shape of a smooth opaque object from one view. Technical report, Cambridge, MA, USA, 1970. 13

[Igarashi1999] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: a sketching interface for 3d freeform design. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99, pages 409–416, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co. 14, 22

[Jackson1998] John D. Jackson. *Classical Electrodynamics*. Wiley, third edition, August 1998. 16

[Jarosz2008] Wojciech Jarosz. *Efficient Monte Carlo Methods for Light Transport in Scattering Media*. PhD thesis, UC San Diego, September 2008. 16

[Johnson2007] M.K. Johnson and H. Farid. Exposing digital forgeries in complex lighting environments. *Information Forensics and Security, IEEE Transactions on*, 2(3):450–461, 2007. 39

[Joshi2008] Pushkar Joshi and Nathan A. Carr. Repoussé: Automatic inflation of 2d artwork. In *SBM*, pages 49–55, 2008. 14, 26, 59

[Kajiya1986] James T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM. 3

[Karsch2011] Kevin Karsch, Varsha Hedau, David Forsyth, and Derek Hoiem. Rendering synthetic objects into legacy photographs. In *Proceedings of the 2011 SIGGRAPH Asia Conference*, SA '11, pages 157:1–157:12, New York, NY, USA, 2011. ACM. 6, 7, 36

[Khan2006] Erum Arif Khan, Erik Reinhard, Roland W. Fleming, and Heinrich H. Bülthoff. Image-based material editing. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 654–663, New York, NY, USA, 2006. ACM. 7, 12

[Kimmel2003] Ron Kimmel, Michael Elad, Doron Shaked, Renato Keshet, and Irwin Sobel. A variational framework for retinex. *Int. J. Comput. Vision*, 52(1):7–23, April 2003. 12

[Land1971] Edwin H. Land and John J. McCann. Lightness and retinex theory. *J. Opt. Soc. Am.*, 61(1):1–11, Jan 1971. 11, 12

[Langer2000] Heinrich H Langer, Michael Sand Bülthoff. Depth discrimination from shading under diffuse lighting. *Perception*, 29:649660. 7

[LopezMoreno2010] Jorge Lopez-Moreno, Sunil Hadap, Erik Reinhard, and Diego Gutierrez. Compositing images through light source detection. *Computers & Graphics*, 34(6):698 – 707, 2010. 8, 22

[LopezMoreno2013] Jorge Lopez-Moreno, Elena Garces, Sunil Hadap, Erik Reinhard, and Diego Gutierrez. Multiple light source estimation in a single image. *Computer Graphics Forum*, 2013. 8

[Marschner1997] Stephen R. Marschner and Donald P. Greenberg. Inverse Lighting for Photography. In *Color Imaging Conference*, 1997. 9

[Nehab2005] Diego Nehab, Szymon Rusinkiewicz, James Davis, and Ravi Ramamoorthi. Efficiently combining positions and normals for precise 3d geometry. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 536–543, New York, NY, USA, 2005. ACM. 14, 59, 60

[Nocedal2006] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, 2nd edition, 2006. 33

[Oh2001] Byong Mok Oh, Max Chen, Julie Dorsey, and Frédo Durand. Image-based modeling and photo editing. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 433–442, New York, NY, USA, 2001. ACM. 12

[Poynton2013] Charles Poynton and Brian Funt. Perceptual uniformity in digital image representation and display. *Color Research & Application*, pages n/a–n/a, 2013. 19

[Ramamoorthi2001a] Ravi Ramamoorthi and Pat Hanrahan. A signal-processing framework for inverse rendering. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 117–128, New York, NY, USA, 2001. ACM. 4, 9, 13, 23, 30, 43

[Ramamoorthi2001b] Ravi Ramamoorthi and Pat Hanrahan. An efficient representation for irradiance environment maps. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 497–500, New York, NY, USA, 2001. ACM. 10

[Rother2004] Carsten Rother, Vladimir Kolmogorov, and Andrew Blake. "GrabCut": interactive foreground extraction using iterated graph cuts. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 309–314, New York, NY, USA, 2004. ACM. 2

[Shewchuk1996] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag,

May 1996. From the First ACM Workshop on Applied Computational Geometry. 26, 68

[Shirley2009] Peter Shirley and Steve Marschner. *Fundamentals of Computer Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 3rd edition, 2009. 19

[Slater2001] Mel Slater, Anthony Steed, and Yiorgos Chrysanthou. *Computer Graphics and Virtual Environments: From Realism to Real - Time*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2001. 19

[Sloan2002] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 527–536, New York, NY, USA, 2002. ACM. 10

[Stroustroup2013] Bjarne Stroustroup. *The C++ Programming Language (4th Edition)*. Addison-Wesley, 2013. 70

[Tappen2005] Marshall F. Tappen, William T. Freeman, and Edward H. Adelson. Recovering intrinsic images from a single image. *IEEE Trans. Pattern Anal. Mach. Intell.*, 27(9):1459–1472, September 2005. 12

[Tomasi1998] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Proceedings of the Sixth International Conference on Computer Vision*, ICCV '98, pages 839–, Washington, DC, USA, 1998. IEEE Computer Society. 12

[Valgaerts2012] Levi Valgaerts, Chenglei Wu, Andrés Bruhn, Hans-Peter Seidel, and Christian Theobalt. Lightweight binocular facial performance capture under uncontrolled lighting. *ACM Trans. Graph.*, 31(6):187:1–187:11, November 2012. 14

[Ward2008] Greg Ward, Erik Reinhard, and Paul Debevec. High dynamic range imaging & image-based lighting. In *ACM SIGGRAPH 2008 classes*, SIGGRAPH '08, pages 27:1–27:137, New York, NY, USA, 2008. ACM. 9

[Weyrich2008] Tim Weyrich, Jason Lawrence, Hendrik Lensch, Szymon Rusinkiewicz, and Todd Zickler. Principles of appearance acquisition and representation. *Foundations and Trends in Computer Graphics and Vision*, 4(2):75–191, 2008. 10

[Wu2011a] Chenglei Wu, B. Wilburn, Y. Matsushita, and C. Theobalt. High-quality shape from multi-view stereo and shading under general illumination. In *Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '11, pages 969–976, Washington, DC, USA, 2011. IEEE Computer Society. 10, 14, 15

[Zhang1999] Ruo Zhang, Ping-Sing Tsai, James Edwin Cryer, and Mubarak Shah. Shape from shading: A survey. *IEEE Trans. Pattern Anal. Mach. Intell.*, 21(8):690–706, August 1999. 13

Chapter 7

Appendix A: Source Code Documentation

The source code that accompanies this final project report is written in C++. It consists of a modular library, and a number of applications that build upon functions and classes from the library project.

The library code implements the relighting pipeline described in section 4. The different command line or GUI applications then share and reuse the code contained in the library. Reuse is facilitated by the libraries' simple design: We avoid complex class hierarchies, but rather provide independent, pure functions. These functions operate on standard data types such as OpenCV images [Bradski2008], OpenMesh triangle meshes [Botsch2002], and Eigen matrices [Guennebaud2010]. Any application which already uses these data types can simply include and use our function library.

There is one exception from this non-object-oriented design: The real-time rendering module contained in the lum::gl namespace (see section 7.0.6). which provides simple means of rendering meshes and images into a three dimensional scene. Here, we use classes, since modeling scene objects like cameras, lights, and meshes is a natural application for object-oriented programming.

7.0.1 lum Namespace Reference

Root Module.

Description

This module contains a mix of library and application code. Library functions in the root module can be very general (such as the geometric transform matrices), or so specific that creating a new namespace would be infeasible. The following documentation will shed light onto the use cases of each function.

The module also contains a set of *controller* classes. These classes contain the core functionality of applications like reshading or SH projection. The controller classes are nor meant to be reused as-is, but serve as example code and starting points for new projects. Application code is described in the next chapter of this manual.

Miscellaneous Functions

- std::map< char, std::string > **parse_cli** (int argc, char **argv, std::string argstring)
- cv::Mat reshade (const cv::Mat &normals, const cv::Mat &reflectance, const Eigen::MatrixXf sh_coeffs)

  High level function that applies Spherical harmonics shading to normals and modulates by material reflectance.
- cv::Mat shadeNormalMap (const cv::Mat &normals, const Eigen::MatrixXf sh_coeffs)

  Subproblem of reshade().
- std::vector< float > ToVector (const Pointset &)

  Turn Eigen Matrix into linearized vector (column major)
- std::string readFile (const std::string &path)

  Generic helper: Read text file.
- Eigen::MatrixXf readMatrix (std::istream &is)

  Read eigen matrix from .m text file.
- float getSSD (const Eigen::MatrixXf &A, const Eigen::MatrixXf &B)

  Sum of squared distances between two matrices.
- Eigen::Vector4d PlaneEquation (const Eigen::Matrix3d &)

  Given three points, define the implicit plane equation defined by those points.
- template<class Vector >
  void **print** (const Vector &v, FILE *f)
- template<class Vector >
  void print (const Vector &v)

  Print STL vector to stdout.
- template<class T >
  Eigen::MatrixXf ToMatrix (const T &v, int n_rows)

  Reshape a vector into (column major) matrix form.
- template<typename T >
  T ToHomgenous (const T &M)

  Append row of ones to a matrix.

- template<typename T >
  void **printMatrix** (const T &M, std::ostream &os)

  Write Matrix (e.g.
- template<typename T >
  void **printMatrix** (const T &M)

  Print matrix to stderr.

Geometric Transformations

This category of functions is useful both for working with individual vertices, and for transforming entire objects in a scenegraph.

For example, the 4x4 affine transformation matrices returned by the functions can be passed to

- The **mesh::Transform()** function in the mesh module. This causes the transformation to be *baked into* the mesh vertices
- Set as the transformation matrix of a Drawable object. The transformation is part of the scene graph, not part of the mesh.
- Passed to shader programs e.g. using the SetUniformByMat4() method of the GLSLProgram class.

- Eigen::Matrix4f **rotateX** (float angle)
- Eigen::Matrix4f **rotateY** (float angle)
- Eigen::Matrix4f **rotateZ** (float angle)
- Eigen::Matrix4f **scaleX** (float sx)
- Eigen::Matrix4f **scaleY** (float sy)
- Eigen::Matrix4f **scaleZ** (float sz)
- Eigen::Matrix4f **scaleXYZ** (float sx, float sy, float sz)
- Eigen::Matrix4f **scaleXYZ** (float s)
- Eigen::Matrix4f **translateX** (float x)
- Eigen::Matrix4f **translateY** (float y)
- Eigen::Matrix4f **translateZ** (float z)
- Eigen::Matrix4f **translateXYZ** (float x, float y, float z)

7.0.2 lum::sh Namespace Reference

Spherical Harmonics Module.

Description

This module is a collection of independent functions. The module is deliberately *non-object-oriented* . Rather than implementing complex class hierarchies, to goal is to minimize interdependencies and to provide a number of useful tools that can be used in all types of projects that need to compute spherical harmonics.

The majority of the functions is about evaluating SH functions. There are also utility functions that for example compute the number of SH coefficients at a given level. Finally, there is a set of higher-level functions that interface with the mesh module. For example, those functions return each vertex normal in spherical coordinates, or set vertex colors according to SH coefficients.

Coordinate System Transformations

Functions and Data Types that convert from cartesian to spherical coordinates and vice versa.

- using spherical_coord_t = std::pair< float, float >

  Encodes a (phi, theta) == (azimuth elevation) coordinate on a unit sphere surface.
- using cartesian_coord_t = std::pair< float, float >

  Encodes a (x, y) == (column, row) coordinate in the 2D plane.
- using spherical_coord_list_t = std::vector< spherical_coord_t >

  List of spherical coordinates.
- spherical_coord_t makeSpherical (float phi, float theta)

  Construct sphrical_coord_t.
- spherical_coord_t toSphericalDirection (const Eigen::Vector3f &)

  Cartesian direction -> spherical.
- Eigen::Vector3f toCartesianNormal (const spherical_coord_t &)

  Spherical coordinate direction -> cartesian.
- spherical_coord_t to_sphere_domain (spherical_coord_t s)

  Ensure that phi in [0,2pi) and theta in [0, pi].
- void assert_in_sphere_domain (spherical_coord_t s)

  Assert that phi in [0,2pi) and theta in [0, pi].

SH Evaluation

This set of function forms the core of the Spherical Harmonics module.

They implement the recursive definitions of spherical harmonics and the associated legendre polymials. The SH evaluation code is copied (with slight modification) from a project from the University of Toronto.

There are versions for real and complex valued Spherical Harmonics. See the main part of this report for a detailled explanation of when we want to use real, and when we want the complex valued version.

Some of the functions just return the function value, others also return gradient information. Gradients are for example useful in optimization procedures and are in fact use to refine the normal the shape normals.

- std::complex< float > **sphericalHarmonic** (int l, int m, const **spherical_coord_t** &)

  Evaluate Spherical Harmonic at given direction.
- float **sphericalHarmonic_r** (int l, int m, const **spherical_coord_t** &)

  Evaluate Real Spherical Harmonic at given direction.
- Eigen::VectorXf **shUptoLevel** (const **spherical_coord_t** &, int upto_level)

  It is efficient to compute more all level at once - can reuse legendre plynomials.
- Eigen::MatrixXf **shGradientUptoLevel** (const **spherical_coord_t** &, int upto_level)

  2 column matrix with gradient in phi and theta direction.
- Eigen::MatrixXf **shAndGradientUptoLevel** (const **spherical_coord_t** &, int upto_level)

  3 column matrix with [real_sh, grad_phi, grad_theta].
- Eigen::MatrixXf **systemMatrix** (const **spherical_coord_list_t** &n_list, int upto_level)

  Evaluate real SHs for a list of vertices.

SH Coefficient Utilities

Spherical harmonics coefficients to not just have a single index.

They are first indexed by level (or order) *l* and then by a second index m in range [-l, l]. The level corresponds to the frequency of the basis function, m corresponds to phase. This category contains functions that provide information about and convert between different notations for the coefficiens.

- int **numSHCoefficientsAt** (int level)

  Computes pow(level+1, 2).
- int **linearIndex** (int l, int m)

  Return linear index for a SH index pair.
- int **levelForNumberofCoefficients** (int num_coeffs)

  Returns sqrt(num_coeffs) - 1.

Project from Image

Useful set of functions to project an equirectangular mapping onto a spherical harmonics basis.

There are also functions that do the inverse (rasterSH family). Those functions take a set of SH coefficients and a target image size as input, and raster an equirectangular mapping from the SH representation.

- Eigen::VectorXf **fromLatLonMap** (const cv::Mat &image, int upto_level)

  Compute orthogonal projection of Latitude Longitude map onto SH basis.
- void **rasterizeSH** (cv::Mat &outimage, const Eigen::VectorXf coeffs, int w=0, int h=0)

  *Inverse of **fromLatLonMap()***
- void **rasterizeSH3Ch** (cv::Mat &outimage, const Eigen::MatrixXf coeffs, int w, int h)

  *BGR version of **rasterizeSH()***

- void rasterizeSH (cv::Mat &outimage, int l, int m, int w=0, int h=0)

  Latitude Longitude Map of a single SH basis function.
- void sampleLatLonGradientMap (cv::Mat &outimage, const Eigen::VectorXf coeffs, int w=0, int h=0)

  Computa a SH gradient map with equirectangular mapping.
- spherical_coord_t latLonMap (float x, float y, float w, float h)

  Convert from (x,y) to (phi,theta) domain.
- cartesian_coord_t inverseLatLonMap (const spherical_coord_t &, float w, float h)

  Convert from (phi,theta) to (x, y) domain.
- float patchArea (int n_phi, int n_theta, float theta)

  Helper for numerical sphere integration from equirectangular mapping.
- float latLonMapIntegral (const cv::Mat &image)

  Helper for numerical sphere integration from equirectangular mapping.

Meshes and Spherical Coordinates

A set of functions that extracts all vertex normals of a mesh in spherical coordinates.

- void colorHarmonic (mesh::MyMesh *mesh, int l, int m)

  Set the vertex color attribute of each vertex according to the normal direction on the unit sphere.
- spherical_coord_list_t normalCoordinates (const mesh::MyMesh &mesh)

  Like normalCoordinates() but returns normal at each vertex.
- spherical_coord_t normalCoordinate (const mesh::MyMesh &mesh, const mesh::MyMesh::VertexHandle &h)

  Normal angle at vertex in spherical coordinates.

Optimization

Functions useful for optimzing SH functions.

The basic minimize() function is not particularly useful since we usually don't simply look for the minimum of a function expressed in the SH basis.

More important are the minimizeXChannel() family of functions that also take a target intensity. These functions are used to minimize the distance between the observed image and the re-rendered image.

minimizeSmooth() is a special case of these functions that sets a smoothness prior depending on neighboring vertices.

- bool minimize3Channel (const Eigen::MatrixXf &coeffs, const Eigen::Vector3f &target, const spherical_coord_t &x0, const spherical_coord_t &x_prior, float prior_weight, spherical_coord_t *x_star)

  Not implemented.
- bool minimize3ChannelList (const Eigen::MatrixXf &coeffs, const Eigen::MatrixXf &target, const spherical_coord_list_t &x0, spherical_coord_list_t *x_star)

Minimize a 3Channel list of vertices without smoothness constraints.

- bool **minimize1Channel** (const Eigen::VectorXf &coeffs, const float &target, const **spherical_coord_t** &x0, const **spherical_coord_t** &x_prior, float prior_weight, **spherical_coord_t** *x_star)

  This is the main minimization procedure whithout smoothness prior.

- bool **minimizeList** (const Eigen::VectorXf &coeffs, const Eigen::VectorXf &target, const **spherical_coord_list_t** &x0, **spherical_coord_list_t** *x_star)

  Minimize a list of vertices without smoothness constraints.

- bool **minimizeSmooth** (const Eigen::VectorXf &coeffs, const Eigen::VectorXf &target, const std::vector< std::vector< size_t >> &connectivity, const **spherical_coord_list_t** &x0, **spherical_coord_list_t** *x_star)

  Minimize with smoothness prior.

- bool **minimize** (const Eigen::VectorXf &coeffs, const **spherical_coord_t** &x0, **spherical_coord_t** *x_star, std::vector< **spherical_coord_t** > *path)

  Minimize a function given in real SH basis.

- template<typename T >
  T **cos_angle** (T x0_phi, T x0_theta, T phi, T theta)

  Helper used in panalty terms.

7.0.3 lum::mesh Namespace Reference

Mesh processing module based on OpenMesh and Eigen.

Description

Some functions in this module perform complex algorithms, such as Bilateral Filtering, shape inflation, of "shape from normals" [Nehab2005]. Others are simple utility functions, for example for mesh scaling or rotation. See below for a detailed description of the different categories.

All functions in this module follow the same convention: The mesh is either passed as a constant reference (const & MyMesh) or as a non-const pointer (MyMesh $*$). Whenever the mesh is passed by reference, the caller can be sure that the mesh will not be mutated by the functions. Whenever a pointer is passed, the mesh *will* be mutatet.

Miscellaneous Functions

- Eigen::MatrixXf **lookupShading** (const **MyMesh** &mesh, const cv::Mat &image)

  Looks up vertex attributes from an image.

Bilateral Filtering

Implementation of the Bilateral Filter for meshes

[Fleishman2003]}. There are two functions, the actual filter and a test function that simulates sensor noise of a range scanner. The **bilateralFilteredCopy()** functions does not mutate the mesh and returns a newly allocated mesh with filtered positions instead.

- void **addNoiseAlongNormal** (**MyMesh** $*$, float sigma)

  Simulate measurement noise: Normally distributed along the vertex normal.
- **MyMesh** $*$ **bilateralFilteredCopy** (const **MyMesh** &mesh, float sigma_domain, float sigma_range)

  Return a smoothed version of the positions argument.

Mesh Inflation

Strongly inspired by "Repousse" [Joshi2008] and "Variational Mesh Editing" by [Botsch2004].

This category contains only a single function so far. The function pins the mesh's boundary vertices to y=0 and inflates all other vertices using a constant curvature.

- void **inflate_mesh** (**MyMesh** $*$, float curvature)

  Inflate a mesh.

Modify Vertex Positions

The key function in this category is **updateVertexPositions()** .

It is an implementation of the last-squared shape-from-shading algorithm described in [Nehab2005]. The function is implemented using sparse matrix algorithms from the Eigen package.

flattenVertexPositions() is a useful helper function to put inflated (see inflate()) or repositioned vertices back into the image plane y=0.

- void updateVertexPositions (MyMesh *mesh, const std::map< size_t, float > &boundary_conditions)

  The method adjust the vertices y position to best explain normal directions.
- void flattenVertexPositions (MyMesh *mesh)

  Simple debug helper.

Vertex Attribute Buffers Used in OpenGL Rendering

This category of functions gives access to vertex attributes such as position, normal, texture coordinates, or color.

The attributes are returned either as a matrix or as a std::vector. There further are ToMatrix() and ToVector() helper functions that translate from one to the other.

One interesting family of functions are the GetXYZFlat() functions. Usually, a single vertex is shared between all adjacent triangles, as this is OpenGLs default mode of operation for smoothly shaded meshes. However, sometimes we want to render a mesh a flat shaded. In this case, we can use the GetXZYFlat() functions to attribute buffers where vertices are duplicated and hence not shared between triangles.

- long GetNumOfVertices (const MyMesh &mesh)

  Number of vertices.
- Pointset GetVertices (const MyMesh &)

  Vertex Positions as Matrix.
- Pointset GetNormals (const MyMesh &)

  Vertex Normals as Matrix.
- Pointset GetVerticesFlat (const MyMesh &)

  Mesh vertices used for flat shading.
- Pointset GetTexCoords (const MyMesh &)

  Texture coordinates as 2x<n_vertices> matrix.
- Pointset GetTexCoordsFlat (const MyMesh &mesh)

  Texture coordinates for flat shading.
- Pointset GetNormalsFlat (const MyMesh &mesh)

  Vertex normals used for flat shading.
- std::vector< unsigned int > GetTriangleIndicesFlat (const MyMesh &)

  Vertex indices for flat shaded geometry.
- std::vector< unsigned int > GetTriangleIndices (const MyMesh &)

  Vertex indices of each face in vector of size 3<n_faces>*
- void SetNormals (MyMesh *mesh, const Pointset &normals)

  Replace normals of the mesh.
- Eigen::Matrix3f GetFaceVertices (const MyMesh &, const MyMesh::FaceHandle &)

3x3 vertex positions of a face.

- std::vector< float > **GetColors** (const **MyMesh** &)

  Vertex RGBA color as floating point vector.

Geometry Transformations

This set of functions transforms the vertex positions and if necessary the normals of mesh.

The most general function of the bunch is **Transform()** which takes a generic 4x4 matrix as argument. There are also more specialized functions for Translation, Rotation, and Scaling.

Other functions need only the mesh as input and bring it to a canonical form. For example, **Normalize()** scales and translates the mesh to be in the y = [-1, 1] range. See the documentation of each individual function for more details.

- Eigen::Vector2f **GetBoundingBox** (const **MyMesh** &mesh)

  Return minimum and maximum y value.
- void **Center** (**MyMesh** *mesh)

  Move mesh so that center of mass is at (0, 0, 0)
- void **Normalize** (**MyMesh** *mesh)

  Uniformly scale mesh so span y = [-1, 1].
- Eigen::Vector3f **GetCentroid** (const **MyMesh** &mesh)

  Center of Mass.
- void **Scale** (**MyMesh** *mesh, float s)

  Scale vertex positions uniformly.
- void **Scale** (**MyMesh** *mesh, const Eigen::Vector3f &v)

  Scale vertex positions nonuniformly.
- void **Add** (**MyMesh** *mesh, const Eigen::Vector3f &v)

  Add offset to all vertices.
- void **Substract** (**MyMesh** *mesh, const Eigen::Vector3f &v)

  Substract offset from all vertices.
- void **Transform** (**MyMesh** *mesh, const Eigen::Matrix4f &)

  Apply transofmration to vertex positions and recompute normals.
- void **RotateX** (**MyMesh** *mesh, float angle)

  Apply roation to vertices.
- std::pair< Eigen::Vector3f,
  Eigen::Vector3f > **AxisAlignedBoundingBox** (const **MyMesh** &)

  Axis Aligned Bounding Box.
- void **ToFirstQuadrant** (**MyMesh** *m)

  Nonuniformly transform mesh such that each of x, y, z spans [0, 1].

Mesh Editing

Functions that merge two meshes into one.

The functions make sure that there are no conflicting indices, but they don't check for intersections or other geometrical aspects. Merge takes to input meshes and returns a new object. Cat() does not create a new object, but instead appends the geometry contained in mesh b to mesh A.

- void Cat (MyMesh *base, const MyMesh &b)

  Append all vertices and triangles of mesh b to existing mesh.
- MyMesh Merge (const MyMesh &a, const MyMesh &b)

  Merge two meshes into one.

Neighborhood Queries

This set of functions is useful for querying information about the neighborhood (adjacent edges, vertices, or faces) of a vertex.

OpenMesh is a half-edge data structure and was specifically designed to allow efficient neighborhood lookups. Here, we add a useful layer on top of OpenMesh's functionality and for example provide access to the intices of neighbor elements in STL library containers.

- int NumAdjacentFaces (const MyMesh &mesh, const MyMesh::VertexHandle &h)

  Valence of the Vertex.
- std::vector< long > VertexNeighbors (const MyMesh &mesh, const MyMesh::VertexHandle &)

  List of indices of neighbor vertices.
- std::vector< long > VertexNeighborEdges (const MyMesh &mesh, const MyMesh::VertexHandle &)

  List of indices of neighbor edges.
- long NumBoundary1RingVertices (const MyMesh &m)
- float NeighborhoodArea (const MyMesh &mesh, const MyMesh::VertexHandle &)

  Compute neighborhood size.

Boundary Queries

Some algorithms like for example mesh inflation treat boundary vertices different than non-boundary vertices.

The functions in this category allow for efficient retrieval and test of boundary vertices

- std::vector< size_t > GetBoundaryVertices (const MyMesh &m)

  Query indices of boundary vertices.
- long NumBoundaryVertices (const MyMesh &m)

  Query number of boundary vertices.
- bool is_in_boundary_onering (const MyMesh &m, const MyMesh::VertexHandle &vh)

  Check if a vertex ...

Access to Individual Elements by Index

Sometimes users need access to just a single vertex attribute.

These functions provide this access in constant time and return the data as Eigen vectors and matrices. GetFace() returns three vertex positions at once. The returned 3x3 matrix can for example be used to establish the plane equation of the polygon.

See Also

PlaneEquation().

- Pointset GetFace (const MyMesh &mesh, const MyMesh::EdgeHandle &h, int v_idx)
  Returns empty matrix if face does not exist (boundary edge)
- Eigen::Vector3f GetVertex (const MyMesh &mesh, long v_idx)
  Query vertex position by index.
- Eigen::Vector3f GetVNormal (const MyMesh &mesh, long v_idx)
  Query vertex normal by index.
- Eigen::Vector3f GetFNormal (const MyMesh &mesh, long v_idx)
  Query face normal by index.
- Eigen::Vector3f GetFaceCentroid (const MyMesh &mesh, long f_idx)
  Query face centroid by index.
- template<typename T >
  Eigen::Vector3f ToEigen (T t)
  Convert from MyMesh::Point to Eigen vectors.
- template<typename T >
  MyMesh::Point ToMeshPoint (T t)
  Convert from Eigen Vector to MyMesh::Point.

Laplacian Operator Utilities

The Laplacian Operator plays an important role in laplacian and spectral mesh editing techniques.

There are various definitions of the operator, for example the uniformly weighted Laplacian or the cotangent-weighted Laplacian. We are on a middle ground: All neighboring vertices are weighted equally, but we achieve scale invariance by weighting the result by the neighborhood area.

We use the laplacian operator for example in the inflate() function.

- float FaceArea (const MyMesh &mesh, const MyMesh::FaceHandle &)
  Used in Vertex Neighborhood Area computation NeighborhoodArea()
- std::map< unsigned, float > UniformLaplaceBeltramiWeights (const MyMesh &, const MyMesh-::VertexHandle &)
  Weights of Uniform Laplace Operator.
- Eigen::Vector3f UniformLaplaceBeltrami (const MyMesh &, const MyMesh::VertexHandle &)
  Compute Uniform Laplace Operator weighted by vertex neighborhood size.
- std::vector< std::vector
  < size_t > > ConnectivityList (const MyMesh &m)
  Return mesh topology as nested list.

Read Mesh from Triangle List

We often construct meshes programmatically or read them from an *indexed face list* file format.

The functions in this category can be used to convert such index lists to a MyMesh object.

In case the data is in a standard file on disk (like .off, .stl, obj), the ReadMesh() function does it all.

- MyMesh $*$ MeshFromTriangleList (const std::vector< Eigen::Vector3f > &vertices, const std::vector< int > &indices)

  *Variation of **MeshFromTriangleList()** with integer index list.*
- MyMesh $*$ MeshFromTriangleList (const std::vector< Eigen::Vector3f > &coords, const std::vector< size_t > &indices)

  Construct MyMesh structure from vertex positions and triangle indices.

Read Mesh from File

- MyMesh $*$ ReadMesh (const std::string &filename)

  Read Mesh file into OpenMesh structure and initialize all required field like vertex normals, colors, etc.

7.0.4 IcoSphere Class Reference

A numerically robust (subdivision) sphere class based on Eigen code.

Description

The most straightforward procedural code for generating sphere meshes works by putting vertices at discretized azimuth and elevation angles. The more fine-grained the discretization, the smoother the sphere becomes. However, the just described discretization produces poorly shaped triangles near the poles. The method is thus unsuitable for geometry processing applications where well-shaped triangles (equilateral in the best case) are desired.

More rubust methods are based on subdivision. This particular implementation starts out with an Icosahedron (12 vertices) at level 0. At each level, the meshe's triangles are subdivided, causing the vertex count to increase by four. Hence, the number of vertices of the generated sphere mesh is $12 \cdot 4^l$. More about Icosahedrons and other Platonic Solids can be found for example in work by Blinn [**Blinn1996**, $Ch.4$].

7.0.5 lum::img Namespace Reference

Image Processing and Computer Vision Module.

Description

Functions in this module read, write, or manipulate 2D image data. Most functions depend on OpenCV, some accept or return Eigen matrices as images.

The module is closely related to the SH module and the mesh module. For example, the SH module has functions that turn a set of SH coefficients into an image by creating an equirectangular mapping of the sphere. The lookupShading() function on the other hand is closely related to to mesh processing. After the foreground is inflated, the mesh is projected back onto the image plane, and its target shading value is looked up. See the function's documentation for details.

Miscellaneous Functions

- cv::Mat **readImageFromVar** (const matvar_t *var)

  Read openCV image from matvar variable.
- mat_t * **openOrCreateMatfile** (const std::string &file)

  Internal helper.
- mat_t * **createMatfile** (const std::string &file)

  Will overwrite any existing file.

Boundary Extraction

"Moore Boundray Tracking" as described in *Gonzalez&Woods* [Gonzalez2006].

extractBoundary() is the main function of this group. It accepts a binary mask and returns a polygon (list of points). This polygon representation can be written to disk using the **writePolyFile()** function from the triangulation category.

OpenCV has similar functionality built-in with the cv::findContours() function. See demoExtractBoundary() for sample code on how to use the OpenCV boundary extractor.

- std::vector< cv::Point > **extractBoundary** (const cv::Mat &mask)

  "Moore Boundray Tracking" See Gonzalez&Woods 3rd ed p.796
- cv::Point **uppermostLeftmost** (const cv::Mat &mask)

  helper for step 1 of the algorithm (see Gonzalez&Woods)
- std::pair< cv::Point, cv::Point > **nextNeighborCW** (const cv::Mat &mask, const cv::Point &current, const cv::Point &start_bg)

  helper for step 3 of the algorithm
- bool **isInBounds** (const cv::Mat &image, const cv::Point &pt)

  Simple helper - name says it all.
- bool **isInBounds** (const cv::Mat &image, int x, int y)

  Simple helper - name says it all.

Read EXR files

A set of functions to read EXR files.

EXR files support HDR images. They are often a good choice for saving intermediary or final results of relighting computations. OpenCV has an EXR reader built in. Unfortunately, the OpenCV function crashed for me, so I had to integrate the EXR SDK directly.

- cv::Mat **readExr** (const std::string &file, bool alpha=true)

  Return the Full Sized image as CV_32UC(3 or 4)
- cv::Mat **readExrPreviewImage** (const std::string &file)

  Return the Preview Image (if available) as CV_8UC4.
- void **printExrAttributes** (const std::string &)

  Print attributes to stdout.

Read and Write .mat Files

This is a category of functions that read and write MATLAB compatible .mat files.

Some functions accept OpenCV objects (greyscale and three channel), others accept Eigen matrices and interpret them as greyscale. See the detailed discussion for more information how RGB files are mapped to OpenCV's BGR format.

- cv::Mat **readImageFromMat** (const std::string &file, const std::string &var)

  Read image from .mat file as CV_64F[C3].
- Eigen::MatrixXf **fromOpenCV** (const cv::Mat &m)

  Convert single channel OpenCV image to Eigen Matrix.
- Eigen::MatrixXf **readMatrixFromVarFile** (const std::string &file, const std::string &var)

  Read .mat file into Eigen matrix.
- void **writeImageToMat** (const cv::Mat &, const std::string &file, const std::string &var)

  Write out greyscale or BGR OpenCV image to .mat file.
- void **writeToMat** (const Eigen::MatrixXf &m, const std::string &file, const std::string &var)

  Write Eigen matrix to .mat file.

Read and Write Images

This is a set of IO functions for image data.

Some simply read standard file formats from disk - functionality provided by OpenCV's imread(). Others are more specific, for example **readFramebuffer()** with reads RGB datafrom the currently bound OpenGL framebuffer. One important set of IO functins reads and writes double precision floating points to the uncompressed .mat format. This format can be read my MATLAB and does not provide any clamping, normalization, or color management as many other image formats do. Henc, .mat files are our preferred format to exchange and store intermediary results.

Once all computations are done, we write the image to a standard output format. The simplest format here is .png (OpenCV supports both 8 and 16 bit .pngs). If we have HDR data that is not adequatly represented with 16bit integers, we can also write to ILM's .exr format.

The IO module depends on the OpenGL, Matio, OpenCV, and OpenEXR libraries.

- std::string **getImageType** (int number)
- cv::Mat readMask (const std::string &)

  Read file from disk and threshold to binary mask.
- cv::Mat read32FC3 (const std::string &, float gamma=1.0)

  Read 3 channel floating point with optional gamma adjustment.
- cv::Mat read32FC1 (const std::string &, float gamma=1.0)

  Read 1 channel floating point with optional gamma adjustment.
- cv::Mat readFramebuffer ()

  Read the currently bound OpenGL framebuffer into an OpenCV image.

Linear Interpolation

For some reason, OpenCV does not support interpolated lookup at non-integer values.

Other image processing packages (for example MATLAB's interp2()) support this very useful function.

This category contains a templated linear interpolation that is compatible with both floating point and integer-valued pixels.

- template<typename T >
  T atSubpixel (const cv::Mat &img, const cv::Point2f &pt)

  Templated Linear Interpolation.
- template<typename T >
  T **atSubpixel** (const cv::Mat &img, float x, float y)

Conversion

Straightforward helper functions: Convert Eigen::Vector3f objects to the corresponding cv::Vec3f type and vice versa.

The type conversions to not attempt any conversion between BGR and RGB data so there is a swizzle function for each data type.

- Eigen::Vector3f toEigen (const cv::Vec3f &v)

  Convert from Eigen to OpenCV Vector type.
- cv::Vec3f toOpenCv (const Eigen::Vector3f &v)

  Convert from OpenCV to Eigen Vector type.
- cv::Vec3f swappedRgbBgr (const cv::Vec3f &v)

  Return copy with swapped Red and Blue channel.
- Eigen::Vector3f swappedRgbBgr (const Eigen::Vector3f &v)

  Return copy with swapped Red and Blue channel.

OpenCV Examples.

Example code for OpenCV.

These functions are not meant to be directly reusable. They simply select paramters for built-in OpenCV functions, document the choices, and display the results. Some of the code examples are inspired by [OpenCV Cookbook 2011].

- void **demoShowRGB** (const cv::Mat &image)
- void **demoIteratePixel** (const cv::InputArray &image)
- void **demoShowHist** (const cv::Mat &image)
- void **demoHisteq** (const cv::Mat &image)
- void **demoLutDemo** (const cv::Mat &image)
- void **demoShowHue** (const cv::Mat &image)
- void **demoMeanShiftFilter** (const cv::Mat &image)
- void **demoMeanShiftChromaFilter** (const cv::Mat &image)
- void **demoMorphological** (const cv::Mat &image)
- void **demoGreyScaleMorphology** (const cv::Mat &image)
- void **demoPrintImageType** (const cv::Mat &image)
- void **demoColormap** (const cv::Mat &image)
- void **demoMouseCallback** (const cv::Mat &image)
- void **demoGauss** (const cv::Mat &image)
- void **demoPyramid** (const cv::Mat &image)
- void **demoFft** (const cv::Mat &image)
- void **demoDistanceTransform** (const cv::Mat &image)
- void **demoExtractBoundary** (const cv::Mat &image)
- void **demoHarris** (const cv::Mat &image)
- void **demoFast** (const cv::Mat &image)
- void **demoSurfDetect** (const cv::Mat &image)
- void **demoRunAll** (const std::string &path)
- void **demoProcessROI** (const cv::Mat &image)
- void **demoSurfDescribe** (const cv::Mat &image, const cv::Mat &image2)

Triangulation

A set of wrapper functions for the Triangle [Shewchuk1996] binary.

Users first extract a polygon they want to triangulate, e.g. using extractBoundary(). This polygon is then written to disk using writePolyFile(). The two runTriangle() functions take this polygon file as input, perform a Constrained Delauney Triangulation in the interior of the polygon, and return the path of the resulting .off file to the caller.

- void writePolyFile (const std::vector< cv::Point > &, std::ostream *)

  See http://www.cs.cmu.edu/~quake/triangle.poly.html *for details on the file format.*
- std::string runTriangle (const std::string &triangle_binary, const std::string &poly_input_file, float max_triangle_area=0.5)

Evoke the triangle binary with a given input file and parameters.

- std::string **runTriangle** (const std::string &triangle_binary, const std::vector< cv::Point > &, float max_triangle_area=0)

  *Combination of **writePolyFile()** and **runTriangle()** for convenience if max_triangle_area is not set, a good default value will be computed s.t.*

7.0.6 lum::gl Namespace Reference

Realtime Rendering Module.

Description

In contrast to the other modules, the rendering code is written in a more object-oriented style. There are classes for cameras, drawable objects, textures, and more. Sometimes these classes are simple structures. The main purpose of these structures is to model the elements of a real scene in an intuitive way. Other classes, like textures or the GLSLProgram class don't model scene objects, but encapsulates calls to the OpenGL 3.2 API. There lies great value in wrapping OpenGL objects in C++ objects. In C programs for example, it is very easy to leak OpenGL resources by forgetting to release them with glDeleteXYZ() calls. In C++, we can use constructur/destructure pairs to ensure that allocated resources are deleted in a deterministic way. See e.g. [Stroustroup2013] for more on resource management in C++.

Beside the classes for scene and OpenGL objects, the module also contains a set of helper functions for working with OpenGL. These functions help defining uniform variables, provide common building blocks such as geometry primitives, or help when the need to debug OpenGL programs arises.

Miscellaneous Functions

- void **raster** (cv::Mat &img, DrawablePtr d)
- Eigen::Matrix4f **projectionMatrix** (float fovy, float aspect, float nearZ, float farZ)

  Returns 4x4 projection matrix.
- Eigen::Matrix4f **orthoMatrix** (float width, float height, float near, float far)

Uniforms and Shader Bindings

A set of functions that configures input and output behaviour of shader programs.

Shader programs run concurrently for each vertex, primitive, or pixel. Many variables, such as vertex attributes, are specified on a per-vertex basis. Uniforms however are variables that are the same for each vertex or pixel. Examples of uniform variables are light positions, light intensity, or the current camera calibration. Some uniforms are concerned with the scene layout, others configure the lighting, and again others specify parameters for the material (BRDF) of the current model. This category provides functions to set the uniform variables for each of those three categories.

If setting uniforms defines the input to the shader programs, *binding locations* is how we define the output behaviour of the program. Binding a location links an output variable name to a numeric framebuffer index. Function bindDefaultLocations() sets the conventional configuration of the GL module. The program object should be re-linked after a call to bindDefaultLocations().

- void **setSceneUniforms** (GLSLProgram *p, const Eigen::Matrix4f &P, const Eigen::Matrix4f &C, const Eigen::Matrix4f &M)

  Set uniforms that describe the scene constellation.
- void **setAppearanceUniforms** (GLSLProgram *program, const Material &material)

  Set uniforms that describe an objects appearance.
- void **bindDefaultLocations** (GLSLProgram *program)

Bind default vertex attribute locations and default fragment output locations.

- void setLightUniforms (GLSLProgram *program, const Eigen::Vector3f &position, const Eigen::Vector3f &color, const float &intensity)

  Set uniforms for a single light source.

OpenGL Information

These functions print information about the current OpenGL context.

Most of the functions in the lum::gl namespace require a *3.2 Core Profile* context so users should always check gl32CoreProfile() before using the module.

- bool gl32CoreProfile ()

  Returns true if we run GL version 3.2 or 3.3.
- std::string PrintOpenGLInfo ()

  Return formatted string with some information about the GL environment.

Geometric Primitives

Not all geometry is from model files.

Geometric primitives like cubes or spheres can be created programatically. This functionality used to be part of the GLUT package. However, GLUT is not compatible with OpenGL 3.2 Core Profile, so this category replicates some of GLUT's functionality. The primitives come as Drawable objects. That is, the geometry has already been stored into Vertex Buffers and Vertex Arrays. The Drawable can be passed be assigned a model transformation and can then be drawn by a Renderer object.

- Drawable * GetUnitCubeDrawable ()

  Primitive Cube Shape.
- void LoadQuadMesh (Drawable *, float w, float h)

  Upload Quadmesh into existing Drawable.
- void **LoadQuadMeshXYQuadrant** (Drawable *m, float w, float h)
- Drawable * GetQuadDrawable (float w, float h)

  Primitive Quad Shape.
- mesh::MyMesh * SphereMesh (unsigned level=5)

  Return an icosphere for a given subdivision level.
- Drawable * GetSphereDrawable (unsigned level)

  Primitive Sphere Shape.

Framebuffer Dump

These functions can be used to dump framebuffer contents into an uncompressed binary floating point file.

The resulting binary file can be parsed and analyzed in another tool, for example MATLAB. Note that there is a similar function readFramebuffer in the img module that returns the framebuffer as an OpenCV image object instead of writing to a binary file.

- std::pair< int, int > getFramebufferSize ()

  Return width and height of the active framebuffer.
- RGBABuffer getColorBuffer ()

  Read color buffer as 32bit float and write to binary file.
- void printDepthBuffer (const std::string &path)

  Read depth as 32bit float and write to binary file.
- void printColorBuffer (const std::string &path)

  Read color buffer as 32bit float and write to binary file.
- std::vector< float > getDepthBuffer ()

  Read depth as 32bit float and write to binary file.

7.0.7 Camera Class Reference

This implements an orbiting camera.

Description

This class represents a camera in the scene graph. The camera accepts a direction, an origin, and a distance from the origin. Controlling these parameters results in an *orbiting* behaviour, where the camera always looks at the same point in space, but circles it with constant distance. Usually, the camera's viewing direction is specified by the user moving the mouse, and the camera distance is controlled by scrolling the mouse wheel.

7.0.8 CamSetup Struct Reference

Builds external camera matrices from (position, lookat) pairs.

Description

A camera is defined by two transformations: one external that defines the camera's position and orientation in the scene, and one internal that specifies the camera's field of view and clipping planes. This setup class helps specifying the external camera configuration. For a helper function to conveniently specify internal parameters see the projectionMatrix() function.

A camera's external configuration exhibits 6 degrees of freedom. Three for the position, three for the camera's orientation. Setting those 6 parameters directly is often unintuitive, and so this class tries to model a photographer's or animator's intuition about camera placement: A camera is placed by specifying it's position, and a second scene point that the camera is to look at. There is one caveat left though: the (position, lookat) pair only specifies a ray the camera is supposed to look along, but leaves the camera's rotation around this ray (the "up" direction) unspecified. The CamSetup class resolves this ambiguity by always putting the camera's up vector towards the positive y-direction in scene coordinates.

7.0.9 Drawable Class Reference

Represents geometry data on the GPU.

Description

This class represents all models that will eventually be drawn by GPU. When a Drawable object is created, it calls OpenGL to create new Vertex Buffer Objects and a new Vertex Array Object. The user can then upload vertex data (positions, colors, normals, texture coordinates) by using one of the UploadXYZ() methods of the Drawable. The Drawable destroys all vertex buffers when it is destroyed, and thus minimizes the risk of resource leaks.

If users which to upload geometry data themselves, for example from an OpenMesh structure, they can create an empty Drawable through the default constructor. Alternatively, the user can acquire a preloaded Drawable from one of the geometry primitive functions such as GetUnitCubeDrawable().

7.0.10 GLSLProgram Class Reference

Resource Management Class for glProgram handle.

Description

An OpenGL 3.2 application is not complete without a vertex shader, a fragment shader, and an optional geometry shader. All shader programs must be compiled individually, and attached to a program object. The program object is then *linked*. In the linking step, two actions are taken: First, it is ensured that input and output variables between the shader stages are declared consistently. Second, numeric vertex attribute (input) and framebuffer (output) values are assigned to the symoblic variables in code. See bindDefaultLocations() for information on this second action in the linking step.

GLSLProgram handles the compilation and linking step in its InitShaderProgram() methods. Any errors during this phase are queried from OpenGL and written to stderr. GLSLProgram further provides methods to upload uniforms in common formats. For example, vectors and matrices can be uploaded as Eigen datatypes.

GLSLProgram encapsulates all resource handling of OpenGL objects. It asks OpenGL to create a program object in its constructor, and tells OpenGL to free the associated resources when the GLSLProgram itself is destroyed. As long as GLSLPrograms are allocated on the stack, or handled through reference-counted pointers, all OpenGL resources are guaranteed to be released properly.

7.0.11 Light Class Reference

Basic struct representing a point light source.

Description

This class encapsulates a point light source for real-time rendering. The light is fully specified by a position in 3D cartesian space, and a RGB triple that determines the light source's hue and intensity.

In relighting, such light sources are not used. Instead, the illumination at each vertex is computed on the CPU. The result is uploaded to the Drawable's color vertex attribute. When we then set the Drawable's ShadingMode to ShadingMode::COLOR, the shader program will pass through the color attribute unmodified.

7.0.12 Material Class Reference

Basic Struct for Phong Model.

Description

This complements the Light struct. It is used to compute the modified phong model [Blinn1977] for real-time rendering on the GPU. The modified phong model is a superpositions of three lighting components: A constant ambient that is independent of any light sources, a diffuse component that is independent of the viewing positions, and a specular component that depends on the position of viewer, object, and light. Each of these components can be controlled by its own set of RGB coefficients. The extent of the specular highlight can further be controlled by a *shininess* parameter, where a high shininess causes the specular highlight to me more intense, but also less spread out.

In order to render a Drawable with phong lighting, the Drawable's ShadingMode must be set to ShadingMode::MATERIAL.

7.0.13 Rasterizer Class Reference

Maintains OpenGL state for orthographic projection, rasterization, and screen space techniques in general.

Description

Drawables are ordered by their z translation where positive z is drawn last. This class is a simplified alternative of the Renderer class for 2D sprite content. It is a good staring point for image processing operations, compositing, or tone mapping. The class is also useful for screen-space rendering techniques, for example screen space ambient occlusion.

In relighting, it is used as a simple rasterizer. Due to the triangulation step, we do not have a one-to-one mapping between vertices and pixels. Hence, whenever we want to store results back into an image file, we must first rasterize our triangle mesh over a pixel grid. The Rasterizer class is very flexible and allows us to rasterize both shading and normals.

7.0.14 Renderer Class Reference

3D OpenGL Renderer

Description

Renders a set of Drawable objects. Each drawable has its own transformation applied. Client classes can controll the camera through the rotateCameraLongitude() and rotateCameraLatitude() functions.

The renderer support multiple rendering modes on a per-drawable basis. See the Drawable's ShadingMode property for details. Also, there are two global rendering modes - wireframe and normals visualization - that can be enabled and disabled by calling toggleDrawWireframe() and toggleDrawNormals() on the Renderer object.

7.0.15 Texture Class Reference

Wrapper for two-dimensional OpenGL textures.

Description

This class encapsulates all interactions with OpenGL's texture pipeline. The class ensures that OpenGL texture objects are created and deleted at the right time. Users can then upload RGB or RGBA image data stored in an OpenCV image object. The Upload() method contains all calls to the OpenGL API and picks reasonable default values, for example for texture filtering.

A call to Use() makes the texture active and binds it to texture unit 0.