

# **An Interactive Shader for Natural Diffraction Gratings**

## **Bachelorarbeit**

der Philosophisch-naturwissenschaftlichen Fakultät  
der Universität Bern

vorgelegt von

Michael Single

2014

Leiter der Arbeit:  
Prof. Dr. Matthias Zwicker  
Institut für Informatik und angewandte Mathematik

## Abstract

In nature color production is the result of physical interaction of light with a surface's nanostructure. In his pioneering work, Stam developed limited reflection models based on wave optics, capturing the effect of diffraction on very regular surface structures. We propose an adaption of his BRDF model such that it can handle complex natural gratings. On top of this, we describe a technique for interactively rendering diffraction effects, as a result of physical interaction of light with biological nanostructures such as snake skins. As input data, our method uses discrete height fields of natural gratings acquired by using atomic force microscopy (AFM). Based on Taylor Series approximation we leverages precomputation to achieve interactive rendering performance (about 5-15 fps). We demonstrate results of our approach using surface nanostructures of different snake species applied on a measured snake geometry. Lastly, we evaluate the qualty of our method by a comparision of the maxima for peak viewing angles using the data produced by our method against the maxima resulting by the grating equation.

# Contents

<b>1</b>	<b>Implementation</b>	<b>1</b>
1.1	Precomputations in Matlab . . . . .	2
1.2	Java Renderer . . . . .	6
1.3	GLSL Diffraction Shader . . . . .	7
1.3.1	Vertex Shader . . . . .	7
1.3.2	Fragment Shader . . . . .	11
1.4	Technical Details . . . . .	17
1.4.1	Texture Lookup . . . . .	17
1.4.2	Texture Blending . . . . .	19
1.4.3	Color Transformation . . . . .	19
1.5	Discussion . . . . .	20
1.5.1	Comparison: per Fragment-vs. per Vertex-Shading . . . . .	20
1.5.2	Optimization of Fragment Shading: NMM Approach . . . . .	20
1.5.3	The PQ Shading Approach . . . . .	22
<b>2</b>	<b>Results</b>	<b>24</b>
2.1	BRDF maps . . . . .	24
2.2	Snake surface geometries . . . . .	33
	<b>List of Tables</b>	<b>38</b>
	<b>List of Figures</b>	<b>38</b>
	<b>List of Algorithms</b>	<b>39</b>
	<b>Bibliography</b>	<b>40</b>

# Chapter 1

## Implementation

In computer graphics, we generally synthesize 2d images from a given 3d scene description<sup>1</sup>. This process is denoted as rendering. A usual computer graphics scene consist of a viewer's eye, modeled by a virtuel camera, light sources and geometries placed in the world<sup>2</sup>, having some material properties<sup>3</sup> assigned to. In our implementation, scene geometries are modeled by triangular meshes for which each triangle is represented by a triple of vertices. Each vertex has a position, a surface normal and a tangent vector associated with.

The process of rendering basically involves a mapping of 3d sceene objects to a 2d image plane and the computation of each image pixel's color according to the provided lighting, viewing and material information of the given scene. These pixel colors are computed in several statges in so called shader programs, directly running on the Graphic Processing Unit (GPU) hardware device. In order to interact with a GPU, for our implementations, we rely on the programing interface of OpenGL<sup>4</sup>, a cross-language, multiplatform API. In OpenGL, there are two fundamental shading pipeline stages, the vertex- and the fragment shading stage, each applied sequentially. Vertex shaders apply all transformations to the mesh vertices and pass this data to the fragment shaders. Fragment shaders receive linearly interpolated vertex data of a particular triangle. They are responsible to compute the color of his triangle.

In this chapter we explain in detail a technique for rendering structural colors due to diffraction effects on natural graings, based on the model we have derived in the previose chapter ??, summarized in section ?? . For this purpose we implemented a reference framework which is based on a class project of the lecture *Computer Graphics* held by Mr. M. Zwicker which I attended in autumn 2012<sup>5</sup>.

For performing the rendering process, our implementation expects being provided by the fol-

---

<sup>1</sup>A usual computer graphics scene consist of a viewer's eye, modeled by a virtuel camera, light sources and geometries placed in the world, having some material properties assigned to.

<sup>2</sup>With the term world we are refering to a global coordinate system which is used in order to place all objects.

<sup>3</sup>Example material properties are: textures, surface colors, reflectance coefficients, refractive indices and so on.

<sup>4</sup>Official website:<http://www.opengl.org/>

<sup>5</sup>The code of underlying reference framework is written in Java and uses JOGL and GLSL<sup>6</sup> in order to comunicate with the GPU and can be found at <https://ilias.unibe.ch/>

<sup>6</sup>JOGL is a Java binding for OpenGL (official website <http://jogamp.org/jogl/www/>) and GLSL is OpenGL's high-level shading language. Further information can be found on wikipedia: [http://de.wikipedia.org/wiki/OpenGL\\_Shading\\_Language](http://de.wikipedia.org/wiki/OpenGL_Shading_Language)

lowing input data<sup>7</sup>:

- the structure of snake skin of different species<sup>8</sup> represented as discrete valued height fields acquired using AFM and stored as grayscale images.
- real measured snake geometry represented as a triangle mesh.

The first processing stage of our implementation is to compute the Fourier Terms of the provided height fields like described in section ???. For this preprocessing purpose we use Matlab relying on its internal, numerically fast, libraries for computing Fourier Transformations<sup>9</sup>. The next stage is to read these precomputed Fourier Terms into our Java renderer. This program also builds our manually defined rendering scene. The last processing stage of our implementation is rendering of the iridescent colorpatterns due to light diffracted on snake skins. We implemented our diffraction model from chapter ??? as OpenGL shaders. Notice that all the necessary computations in order to simulate the effect of diffraction are performed within a fragment shader. This implies that we are modeling pixelwise the effect of diffraction and hence the overall rendering quality and runtime complexity depends on rendering window's resolution.

In the following sections of this chapter we are going to explain all render processing stages in detail. First, we discuss, how our precomputation process, using Matlab, actually works. Then, we introduce our Java Framework. It is followed by the main section of this chapter, the explanation how our OpenGL shaders are implemented. The last section discusses an optimization of our fragment shader such that it will have interactive runtime.

## 1.1 Precomputations in Matlab

Our first task is to precompute the two dimensional discrete Fourier Transformations for a given input height field, representing a natural grating. For that purpose we have written a small Matlab<sup>10</sup> script conceptualized in algorithm 1. Our Matlab script reads a given image, which is representing a nano-scaled height field, and computes its two dimensional DFT (2dDFT) by using Matlab's internal Fast Fourier Transformation (FFT) function, denoted by *ifft2*<sup>11</sup>. Note that we only require one color channel of the input image, since the input image is representing an height field, encoded by just one color. Keep in mind that taking the Fourier transformation of an arbitrary function will result in a complex valued output which implies that we will get a complex value for frequency pairs of our input image. Therefore, for each input image we get as many output images, representing the 2dDFT, as the minimal number of taylor terms required for a well-enough approximation. In order to store our output images, we have to use two color channels instead of just one like it was for the given input image. Some example visualizations for the Fourier Tranformation are shown in figure 1.1. We store these intermediate results as binary files to offer floating point precision for the run-time computations to ensure higher precision.

---

<sup>7</sup>All data is provided by the Laboratory of Artificial and Natural Evolution in Geneva. See their website:[www.lanevol.org](http://www.lanevol.org)

<sup>8</sup>We are using height field data for Elaphe and Xenopeltis snakes individuals like shown in figure ???

<sup>9</sup>Actually we use Matlab's inverse 2d Fast Fourier Transformation (FFT) implementation applied on different powers of quation ???. Further information can be read up in section 1.1

<sup>10</sup>Matlab is a interpreted scripting language which offers a huge collection of mathematical and numerically fast and stable algorithms.

<sup>11</sup>Remember, even we are talking about fourier transformations, in our actual computation, we have to compute the inverse fourier transformation. See paragraph ??? for further information. Furthermore our height fields are two dimensional and thus we have to compute a 2d inverse fourier transformation.

In our script every discrete frequency is normalized by its corresponding DFT extrema<sup>12</sup> in the range [0, 1] and the range extrema are stored separately for each DFT term. The normalization is computed the following way:

$$\begin{aligned} f : [x_{min}, x_{max}] &\rightarrow [0, 1] \\ x \mapsto f(x) &= \frac{x - x_{min}}{x_{max} - x_{min}} \end{aligned} \tag{1.1}$$

Where  $x_{min}$  and  $x_{max}$  denote the extreme values of a DFT term. Later, during the shading process of our implementation, we have to apply the inverse mapping. This is non-linear interpolation which is required in order to rescale all frequency values in the DFT terms.

---

<sup>12</sup>We are talking about the i2dFFT of our height fields to the power of n. This is an N by N matrix (assuming the discrete height field was an N by N image), for which each component is a complex number. Hence, there is a complex extrema as well as a imaginary extrema.

---

**Algorithm 1** Precomputation: Pseudo code to generate Fourier terms

---

**INPUT** *heightfieldImg, maxH, dH, termCnt*

**OUTPUT** *DFT terms stored in Files*

```
% maxH: A floating-point number specifying
%       the value of maximum height of the
%       height-field in MICRONS, where the
%       minimum-height is zero.
%
% dH: A floating-point number specifying
%      the resolution (pixel-size) of the
%      'discrete' height-field in MICRONS.
%      It must be less than 0.1 MICRONS
%      to ensure proper response for
%      visible-range of light spectrum.
%
% termCnt: An integer specifying the number of
%           Taylor series terms to use.

function ComputeFFTImages( heightfieldImg , maxH, dh, termCnt )
dH = dh*1E-6;
% load patch into heightfieldImg
patchImg = heightfieldImg .*maxH;
% rotate patchImg by 90 degrees
for t = 0 : termCnt
    patchFFT = power(1j*patchImg , t );
    fftTerm{t+1} = fftshift( ifft2( patchFFT ) );

    % rescale terms as
    imOut(:,:,1) = real(fftTerm{t+1});
    imOut(:,:,2) = imag(fftTerm{t+1});
    imOut(:,:,3) = 0.5;

    % rotate imOut by -90 degrees
    % find real and imaginary extrema of
    % write imOut, extrema , dH, into files .
end
```

---

The key idea of algorithm 1 is to compute iteratively the Fourier Transformation for different powers of the provided height field. These DFT values are scaled by according to their extrema values. Another note about the command `fftshift`: It rearranges the output of the `ifft2` by moving the zero frequency component to the centre of the image. This simplifies the computation of DFT terms lookup coordinates during rendering.

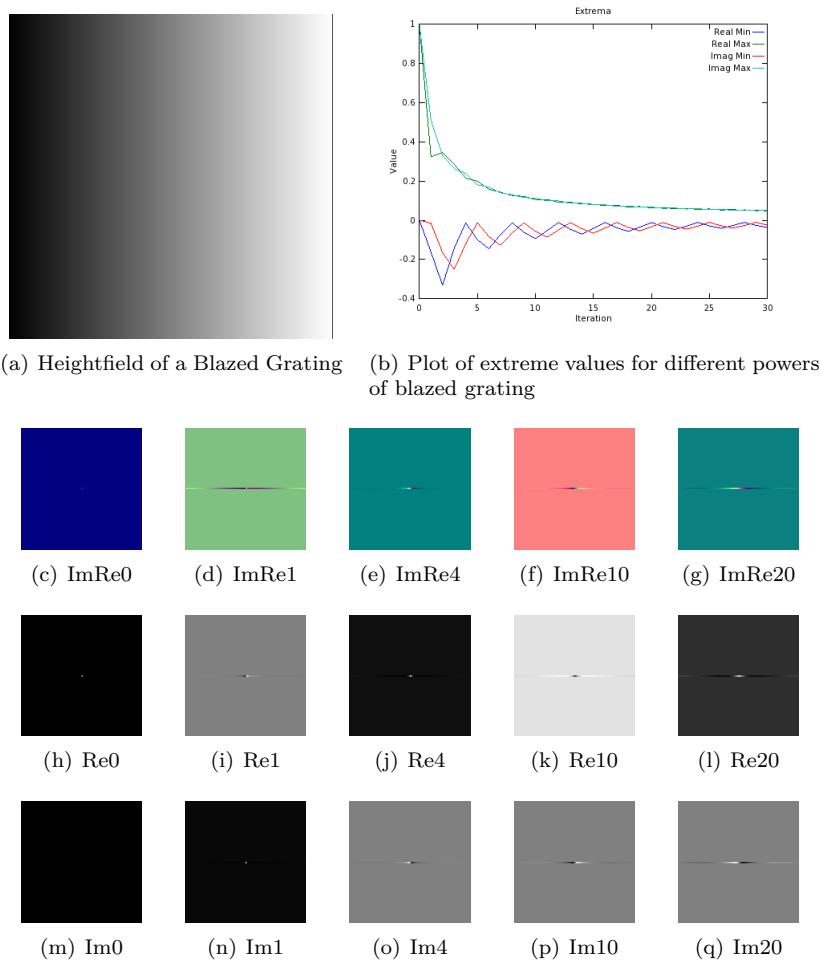


Figure 1.1: A visualization of DFT terms for a height field of a Blazed Grating.

In figure 1.1 we see examples of a visualization of Fourier Transformations generated by our Matlab script for a blazed grating<sup>13</sup> as an input height field image, shown in figure 1.1(a). Figure 1.1(b) shows plots of the extreme values of DFT terms for different powers of the blazed grating. We recognize that, the higher the power of the grating becomes, the closer the extreme values of the corresponding DFT terms get. The figure line from figure 1.1(c) until figure 1.1(g) show us example visualizations of DFT terms for different powers of our grating's height field. Remember that DFT terms are complex valued matrices of dimension as their height field has. In this visualization, all real part values are stored in the red- and the imaginary parts in the green color channel of an DFT image. The figure line from figure 1.1(h) till figure 1.1(l) show us the real part images from above's line corresponding figures. Similarly for the figure line from figure 1.1(m) until figure 1.1(q) showing the corresponding imaginary part DFT term images.

<sup>13</sup>A blazed grating is a height field consisting of ramps, periodically aligned on a given surface.

## 1.2 Java Renderer

This section explains the architecture of the rendering program which I implemented<sup>14</sup> and used for this project. The architecture of the program is divided into two parts: a rendering engine, the so called jrtr (java real time renderer) and an application program. Figure 1.2 outlines the architecture of the renderer.

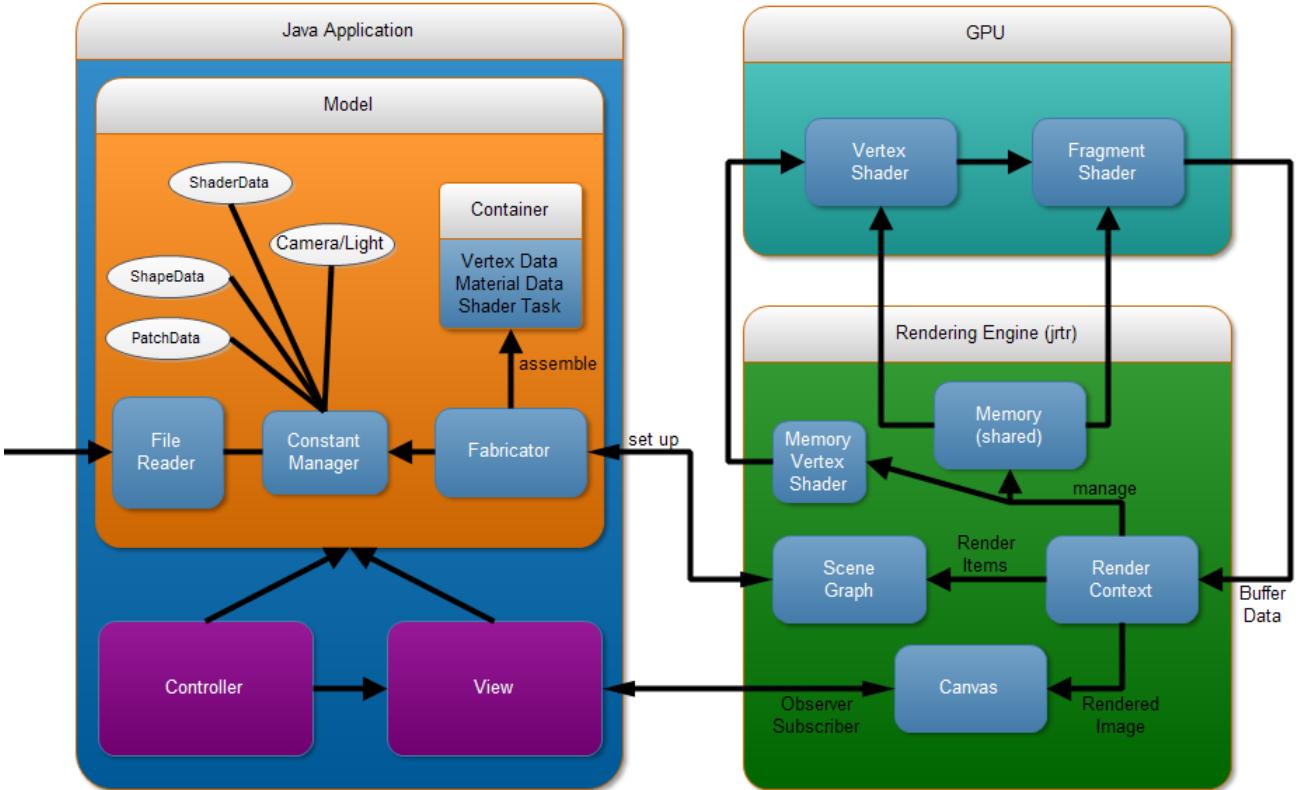


Figure 1.2: Schematical architecture of our Java renderer.

The application program relies on the MVC (Model-View-Controller) architecture pattern. The View just represents a canvas in which the rendered images are shown. The Controller implements the event listening functionalities for interactive rendering within the canvas. The Model of our application program consists of a Fabricator, a file reader and a constants manager. The main purpose of a Fabricator is to set up a rendering scene by accessing a constant manager containing many predefined scene constants. A scene consists of a camera, a light source, a frustum, shapes and their associated material constants. Such materials include a shape texture, precomputed DFT terms<sup>15</sup> for a given height field<sup>16</sup> like visualized in figure 1.1. A shapes is a geometrical object defined by a triangular mesh as shown in figure 1.3.

<sup>14</sup>This program is based on the code of a java real-time renderer, developed as a student project in the computer graphics class, held by M. Zwicker in autumn 2012.

<sup>15</sup>See section 1.1 for further information.

<sup>16</sup>and other height field constants such as the maximal height of its bumps or its pixel real-world width correspondence.

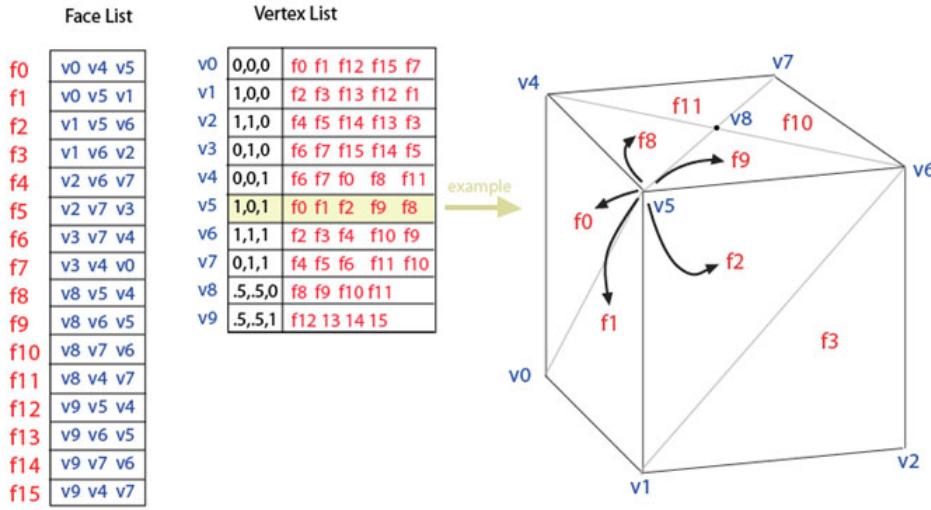


Figure 1.3: Representation<sup>17</sup> of a triangular mesh represents an object as a set of triangles and a set of vertices.

Such a mesh is represented as a data structure consisting of a list of vertices, each stored as a triplet of  $x$ ,  $y$ ,  $z$  positions and triangles, each defined by a triple of vertex-indices. Besides its position, a vertex can have further data assigned to, like a surface color, normals and texture coordinates. The whole scene is encapsulated in a scene graph data-structures, defined and managed within the rendering engine. A scene graph contains all scene geometries and their transformations in a tree like structured hierarchy.

All required configuration, in order to communicate with the GPU through OpenGL, is performed in the jrtr rendering engine. Furthermore, jrtr's render context object, the wholeresource-management for various types of low-level buffers, which are used within the rendering pipeline by our GLSL shaders, takes place in the rendering place. More precisely, this means allocating memory for the buffers, assigning them with scene data and flushing them, when not used anymore. The whole shading process is performed in the GPU, stage-wise: The first stage is the vertex shader (see section 1.3.1) followed by the fragment shader (see section 1.3.2). The jrtr framework also offers the possibility to assign user-defined shaders written in GLSL.

## 1.3 GLSL Diffraction Shader

### 1.3.1 Vertex Shader

In our implementation we want to simulate the structural colors a viewer sees when light diffracted is on grating, e.g. on the skin of a snake. For this purpose, we reproduce a 2d image of a given 3d scene as seen from the perspective of a viewer for given lightning conditions. The color computation of an image is performed in the GPU shaders of the rendering pipeline. In OpenGL, there are two basic shading stages performed to render an image whereas the vertex shader is the first shading stage in the rendering pipeline.

<sup>17</sup>Modified image which originally has been taken from [http://en.wikipedia.org/wiki/Polygon\\_mesh](http://en.wikipedia.org/wiki/Polygon_mesh)

As an input, a vertex shader, like illustrated in figure 1.4, receives one vertex of a mesh and other vertex data such as a vertex normals. It only can access this data and has no information about the neighborhood of a vertex or the topology of its corresponding shape. Since vertex positions of a shape are defined in a local coordinate system<sup>18</sup> and we want to render an image of the perspective of viewer, we have to transform the locally defined positions to a perspectively projected viewer space. Therefore, the main purpose<sup>19</sup> of a vertex shader is to transform the position of vertices. Notice that a vertex shader can manipulate the position of a vertex, but cannot generate additional mesh vertices. Therefore, the output of any vertex shader is a transformed vertex position. Keep in mind that all vertex shader outputs will be used within the fragment shader. For an example, please have a look at our fragment shader 1.3.2.

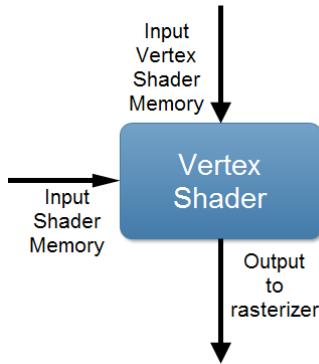


Figure 1.4: Illustration of vertex shader in OpenGL's rendering pipeline.

In the following let us consider the whole transformation, applied in the vertex shader, in depth. Let  $p_{local}$  denote the position of a shape vertex, defined in a local coordinate system. Then the transformation from  $p_{local}$  into the perspective projected position as seen by a observer  $p_{projective}$  looks like the following:

$$p_{projective} = P \cdot C^{-1} \cdot M \cdot p_{local} \quad (1.2)$$

where  $P$ ,  $C^{-1}$  and  $M$  are transformation matrices<sup>20</sup>, defined the following way:

**Model matrix  $M$ :** Each vertex position of a shape is initially defined in a local coordinate system.

To make it feasible to place and transform shapes in a scene, a reference coordinate system, the so called world space, has to be introduced. Hence, for every shape a matrix  $M$  is associated, defining the transformation from its local coordinate system into the world space.

**Camera matrix  $C$ :** A camera models how the eye of a viewer sees an object, defined in world space like shown in figure 1.5. For calculating the transformation matrix  $C$ , a viewer's eye position and viewing direction, each defined in world space, are required. Therefore,  $C$  denotes a transformation from coordinates defined in camera space into the world space. Thus, in order to transform a position from world space to camera space, we have to use the inverse of  $C$ , denoted by  $C^{-1}$ .

<sup>18</sup>Defining the positions of a shape in a local coordinate system simplifies its modeling process and allows us to apply transformations to a shape.

<sup>19</sup>Furthermore, texture coordinates used for texture-lookup within the fragment shader and per vertex lightning can be computed.

<sup>20</sup>These transformation matrices are linear transformations expressed in homogenous coordinates.

**Frustum  $P$ :** The Matrix  $P$  defines a perspective projection onto image plane, i.e. for any given position in camera space,  $P$  determines the corresponding 2d image coordinate. Perspective projections project along rays that converge in center of projection.

Since we are interested in modeling how a viewer sees structural colors on a given scene shape as shown in figure 1.5, modeling a viewer's eye by formulating the corresponding camera matrix  $C$ , is the most important component of the whole transformation series applied in the vertex shader. Hence, we next will have a closer look in how a camera matrix  $C$  actually can be computed.

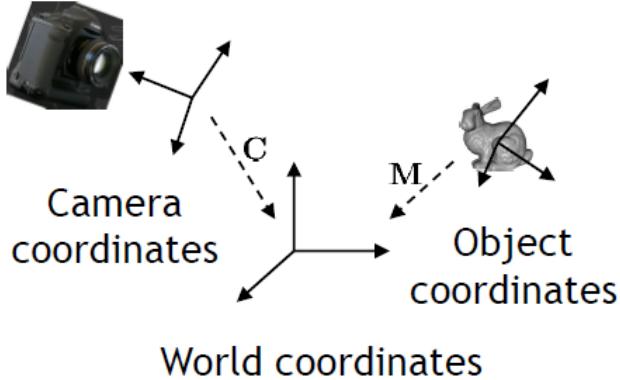


Figure 1.5: Illustration<sup>21</sup> of the Camera coordinate system where its origin defines the center of projection of camera.

The camera matrix  $C$  is constructed from its center of projection  $e$ , the position  $d$  where the camera looks at and a direction vector  $up$ , defining what is the direction in camera space pointing upwards. These components,  $e$ ,  $d$  and  $up$ , are defined in world coordinates. Figure 1.6 illustrates geometrical setup required in order to construct  $C$ .

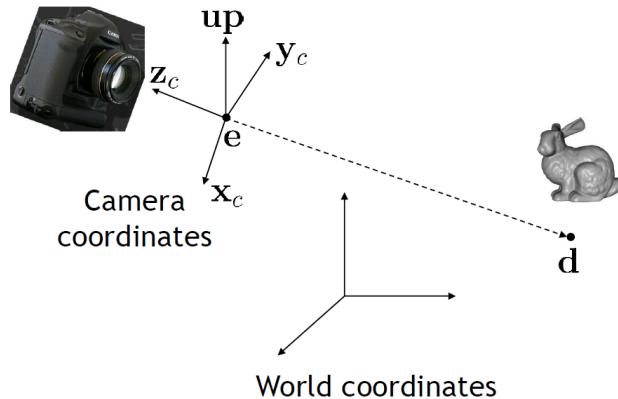


Figure 1.6: Illustration<sup>22</sup> of involved components in order to construct the camera matrix  $C$ . The eye-vector  $e$  denotes the position of the camera in space,  $d$  is the position the camera looks at, and  $up$  denotes the camera's height. The camera space is spanned by the vectors helper vectors  $x_c$ ,  $y_c$  and  $z_c$ . Notice that objects we look at are in front of us, and thus have negative  $z$  values

<sup>21</sup>This image has been taken from the lecture slides of computer graphics class 2012 which can be found on ilias.

The mathematical representation of these vectors,  $x_c$ ,  $y_c$  and  $z_c$ , spanning the camera space, introduced in figure 1.6, looks like the the following:

$$\begin{aligned} z_c &= \frac{e - d}{\|e - d\|} \\ x_c &= \frac{up \times z_c}{\|up \times z_c\|} \\ y_c &= z_c \times x_c \end{aligned} \quad (1.3)$$

As we can see,  $x_c$ ,  $y_c$  and  $z_c$  are independent unit vectors. Therefore, they span a 3d space, the so called camera matrix. In order to express a coordinate in camera space, we have to project it onto these unit vectos. Using a homogenous coordinates representation, this a projection onto these unit vectors can be formulated by the transformation matrix  $C$ :

$$C = \begin{bmatrix} x_c & y_c & z_c & e \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.4)$$

In our vertex shader, besides transforming the vertex positions like described in equation 1.2, for every vertex, we also compute the direction vectors  $\omega_i$  and  $\omega_r$  described like in figure ???. Those direction vectors are transformed onto the tangent space, a local coordinate system spanned by a vertex's normal, tangent and binormal vector. For further information and more insight about the the tangent space, please bave a look at the appendix in the section ???. The algorithmic idea of our vertex shader, stating all its computational steps, is conceptualized in algorithm 2.

---

**Algorithm 2** Vertex diffraction shader pseudo code

---

**Input:** *Mesh* with vertex *normals* and *tangents*  
           Space tranformations  $\{M, C^{-1}, P\}$   
           Light direction *lightDirection*  
**Output:** Incident light and viewer direction  $\omega_i$ ,  $\omega_r$   
           Transformed position  $p_{per}$

**Procedures:** *normalize()*, *span()*, *projectVectorOnTo()*

```

1: Foreach VertexPosition position  $\in$  Mesh do
2:   vec3 N = normalize(M * vec4(normal, 0.0).xyz)
3:   vec3 T = normalize(M * vec4(tangent, 0.0).xyz)
4:   vec3 B = normalize(cross(N, T))
5:   TangentSpace = span(N, T, B)
6:   viewerDir = ((cop_w - position)).xyz
7:   lightDir = normalize(lightDirection)
8:    $\omega_i$  = projectVectorOnTo(lightDir, TangentSpace)
9:    $\omega_r$  = projectVectorOnTo(viewerDir, TangentSpace)
10:  normalize( $\omega_i$ ); normalize( $\omega_r$ )
11:   $p_{per}$  = P · C-1 · M · pobj
12: end for

```

---

<sup>22</sup>This image has been taken from the lecture slides of computer graphics class 2012 which can be found on ilias.

As input, our vertex shader algorithme 2 takes a mesh with of a given scene shape. Each of this vertexc should have a normal and a tangent assigned to. Furthermore, the direction of the scene light is required. For our implementation we always used directional light sources. An example of an directional light source is given in figure 1.7.

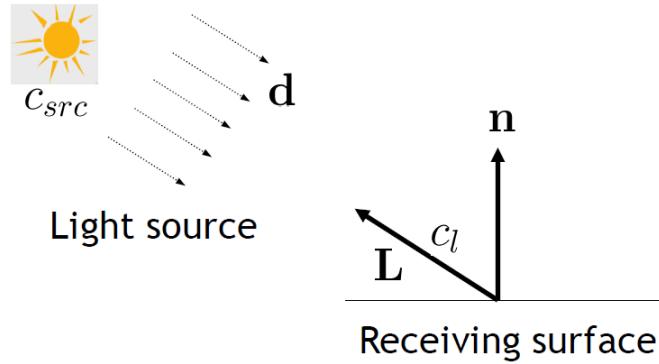


Figure 1.7: Illustration<sup>23</sup> of our light source setup. For a directional light source, all light rays are in parallel.

Last, in order to transform the positions of our mesh like described in equation 1.2, we also have to pass these transformation matrices<sup>24</sup>. For simplification purposes, we introduced the following helper procedures used in the vertex shading algorithm:

**normalize()**: Computes the normalized version of a given input vector.

**span()**: Assembles a matrix from a given set of vectors. This matrix spans a vector space.

**projectVectorOnTo()**: Takes two arguments, a vector and a matrix. The first argument is projected onto each column of a given matrix. And returns a vector living the space spaned by the given argument matrix.

The output of our vertex shader is on the one side the transformed vertex postion and on the other side the incident light  $\omega_i$  and viewing direction  $\omega_r$ , both transformed into the tangent space. The output of the vertex shader is used as the input of the fragment shader, discussed in the next section.

### 1.3.2 Fragment Shader

In the previous section we gave an introduction to the first shading stage of the OpenGL rendering pipeline by explaining the basics of a vertex shaders. Furthermore, we conceptually discussed the idea behind our vertex shading algorithm formulated in algorithm 2. Summarized, the main purpose of our vertex shader is to compute the light- and viewing-direction vectors  $\omega_i$  and  $\omega_r$  defined like in figure ??.

After the vertex-shading stage, the next stage in the OpenGL rendering pipeline is the *rasterization* of mesh triangles. As an input, a rasterizer takes a triple of mesh-triangle spanning vertices,

<sup>23</sup>This image has been taken from the lecture slides of computer graphics class 2012 which can be found on ilias.

<sup>24</sup>When speaking about transformation matrices, we are referring to the model, camera and frustum matrix.

each previously processed by a vertex shader. For each pixel lying inside the current mesh triangle, a rasterizer computes its corresponding position in the triangle. According to its computed position, a pixel also gets interpolated values of the vertices attributes of its mesh triangle assigned. The set of interpolated vertex attributes together with the computed position of a pixel is denoted as a fragment. Figure 1.8 conceptualizes the idea of processed set of fragment computed by a rasterizer.

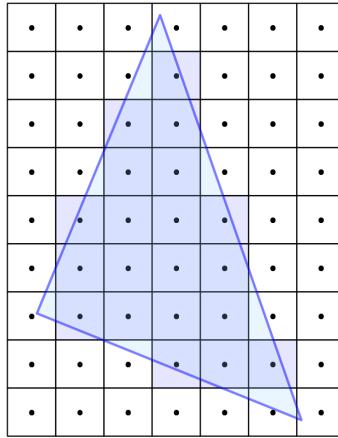


Figure 1.8: Illustration<sup>25</sup> of fragments covered by mesh triangle computed by the OpenGL rasterizer.

A fragment shader as shown in figure 1.9, is the OpenGL pipeline stage subsequent after a mesh triangle is rasterized in the rasterization stage. As input value, a fragment shader takes at least a fragment computed by the rasterizer. It is also possible to assign custom, non-interpolated values to a fragment shader. For each fragment in the fragment shading stage, a color value is computed. Furthermore, a fragment shader computes a depth value for each of its fragments, determining their visibility. Since all existing scene vertices were perspectively projected onto the 2d image plane, the rasterizer could have produced several fragments having assigned the same pixel position. Therefore, among all fragments with the same pixel position, the outputted pixel color is equal to the color of that fragment, which depth is closest to the viewer.

---

<sup>25</sup>This image was taken from [http://en.wikibooks.org/wiki/GLSL\\_Programming/Rasterization](http://en.wikibooks.org/wiki/GLSL_Programming/Rasterization)

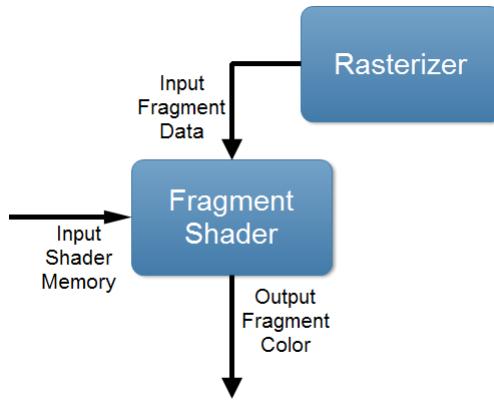


Figure 1.9: Illustration of fragment shader in OpenGL's rendering pipeline.

In this section we explain how to render structural colors resulting due to light diffracted on a natural grating, based on the model described in section ???. The color values of the produced structural colors, resulting from our model, are computed by our fragment shader (later denoted as **FLSS**) which expects being provided by the following input:

- Precompute DFT terms of the provided height field as explained in section 1.1.
- The processed output, produced during the vertex shading stage (see section 1.3.1), which is, the light- and viewing-direction vectors  $\omega_i$  and  $\omega_r$ .
- Fragments produced during the rasterization stage using the output of our vertex shader.

Apart from these basic, varying input values, the following set of shading constants are initialized:

- The number of iterations used for the Taylor series approximation, determining the approximation accuracy.
- The wavelength spectrum  $\Lambda = [\lambda_{min}, \lambda_{max}]$  with a certain discretizing level  $\lambda_{step}$ .
- The color weights of the  $CIE_{XYZ}$  color matching functions.
- The height field image resolution and its pixel to width correspondence.

By using all these inputs, our fragment shader performs a numerical integration over the given wavelength spectrum  $\Lambda$  for our final derived expression, stated in equation ???. For this integration we use the trapezoidal-rule with uniform discretization of the wavelength spectrum at  $\lambda_{step} = 5\text{nm}$  step sizes. This implies we are compressing sampled frequencies to the region near the origin due to the fact we are dividing the  $(u, v)$  by the wavelength  $\lambda$  and this implies that the  $(u, v)$  space is sampled non-linearly.

In section ?? we have seen that we have to multiply our DFT terms by a Gaussian Window in order to approximate the DTFT which our model is based on. This windowing approach is performed for each discrete  $\lambda$  value using a window large enough to span  $4\sigma_f$  in both dimensions. Our DFT terms are computed from height fields that span at least  $65\mu\text{m}^2$  and are sampled at a

resolution of at least  $100\mu m$ . This ensures that the spectral response encompasses all the wavelengths in the visible spectrum.

Next, we will discuss the actual fragment shading algorithm, listed in algorithm 3. Note, that our fragment shading algorithm uses some helper procedures. The two most fundamental one is the **getlookupCoords** which computes the lookup coordinates in the DFT terms for a given  $(u,v)$  defined like in equation ?? and a wavelength  $\lambda$ . The actual computation these coordinates is described in section 1.4.1. Notice that the routine **getLocalLookUp** computes a local lookup coordinates used during the gaussian window approximation explained in section ?? . The routine **distVecFromOriginTo** is used to compute the direction vector from the current position of a fragment in texture space, to one of its texture space living 1-neighborhood neighbors.

---

**Algorithm 3** Fragment diffraction shader pseudo code

---

**Input:** Precomputed DFT Terms  
   Mesh Triangles  
    $\omega_i$  and  $\omega_r$

**Output:** Structural Color of a pixel

**Procedures**<sup>26</sup>:

- getColorWeights*: get colormatching value for wavelength  $\lambda$ (see section ??)
- getlookupCoords*: get lookup coordinate(Eq.1.8) by viewing-& light direction
- distVecFromOriginTo*: get direction vector from origin to a given position
- getLocalLookUp*: get lookup coordinate(Eq.1.9) of DFT windwowing values
- rescaledFourierValueAt*: rescales dft terms according to equation 1.1
- gaussWeightOf*: computes gaussian window according to equation ??
- dot*: computes the dot-product of two given vectors
- gammaCorrect*: apply camma correction on RGB vector(see section 1.4.3)

```

1: Foreach Pixel  $p \in Fragment$  do
2:   INIT  $BRDF_{XYZ}, BRDF_{RGB}$  TO  $vec4(0.0)$ 
3:    $(u, v, w) = -\omega_i - \omega_r$ 
4:   for ( $\lambda = \lambda_{min}; \lambda \leq \lambda_{max}; \lambda = \lambda + \lambda_{step}$ ) do
5:      $xyzWeights = getColorWeights(\lambda)$ 
6:      $lookupCoord = getlookupCoords(u, v, \lambda)$ 
7:     INIT  $P$  TO  $vec2(0.0)$ 
8:      $k = \frac{2\pi}{\lambda}$ 
9:     for ( $n = 0$  TO  $T$ ) do
10:       $taylorScaleF = \frac{(kw)^n}{n!}$ 
11:      INIT  $F_{fft}$  TO  $vec2(0.0)$ 
12:       $anchorX = int(floor(center.x + lookupCoord.x * fftImWidth))$ 
13:       $anchorY = int(floor(center.y + lookupCoord.y * fftImHeight))$ 
14:      for ( $i = (anchorX - winW)$  TO ( $anchorX + winW$ )) do
15:        for ( $j = (anchorY - winW)$  TO ( $anchorY + winW$ )) do
16:           $dist = distVecFromOriginTo(i, j)$ 
17:           $pos = getLocalLookUp(i, j, n)$ 
18:           $fftVal = rescaledFourierValueAt(pos)$ 
19:           $fftVal *= gaussWeightOf(dist)$ 
20:           $F_{fft} += fftVal$ 
21:        end for
22:      end for
23:       $P += taylorScaleF * F_{fft}$ 
24:    end for
25:     $xyzPixelColor += dot(vec3(|P|^2), xyzWeights)$ 
26:  end for
27:   $BRDF_{XYZ} = xyzPixelColor * C(\omega_i, \omega_r) * shadowF$ 
28:   $BRDF_{RGB}.xyz = D_{65} * M_{XYZ-RGB} * BRDF_{XYZ}.xyz$ 
29:   $BRDF_{RGB} = gammaCorrect(BRDF_{RGB})$ 
30: end for

```

---

<sup>26</sup>Please have a look at section 1.3.2 to see further descriptions of these procedures.

Please note that for simplification purposes we omitted some input values in algorithm 3. A complete input value list can be found in section 1.3.2. Last, a brief description of some code sections of our fragment shading algorithm:

#### **From line 4 to 26:**

This loop performs uniform sampling along wavelength spectrum  $\Lambda$  for the spectral integration seen in section ???. The procedure  $ColorWeights(\lambda)$  computes the color weight for the current wavelength  $\lambda$  by a linear interpolation between the color weight and the values  $\lceil \lambda \rceil$  and  $\lfloor \lambda \rfloor$ . The color weights are stored in a external table accessed by our fragment shader<sup>27</sup>. At line 6 the procedure call  $lookupCoord(u, v, \lambda)$  returns the coordinates for the texture lookup which are computed like described in equation 1.7. At Line 25 the diffraction color contribution, computed during the integration over the wavelength spectrum for each wavelength  $\lambda$ , is accumulated.

#### **From line 9 to 24:**

This loop performs the Taylor series approximation using a predefined number of iterations. Basically, the spectral response is approximated for our current value for  $(u, v, \lambda)$ . According to section ???, we can approximate a DTFT by multiplying a gaussian window by DFT terms. When interpreting our DFT terms by a set of matrices, a particular DFT term value, corresponding to the position of a given fragment, can be looked up at the index  $(anchorX, anchorY)$ . For our the Gaussian-Windowing approach we use a one-neighborhood around  $(anchorX, anchorY)$  in order to approximate the DFT value for a fragment.

#### **From line 14 to 22:**

In this inner most loop, the convolution of the gaussian window with the DFT terms of the given height field is performed. The routine  $gaussWeightOf(dist)$  computes the weights in equation (??) from the distance between the current fragment's position and the current neighbor's position in texture space. Local lookup coordinates for the current fourier coefficient  $fftVal$  value, stored in the DFT terms, are computed at line 17 and computed like described in equation 1.9. The actual texture lookup is performed at line 18 using those local coordinates. Inside the procedure  $rescaledFourierValueAt$  the values of  $fftVal$  are rescaled by its extrema values<sup>28</sup>, since  $fftVal$  is normalized according to the description from section 1.1. The current  $fftVal$  value in iteration is scaled by the current Gaussian Window weight and then summed to the final neighborhood FFT contribution at line 20.

#### **After line 26:**

At line 27 the gain factor  $C(\omega_i, \omega_r)$  from equation ?? is multiplied by the computed pixel color like formulated in equation ???. The gain factor contains the geometric term of equation ?? and the Fresnel term  $F$ . For computing the Fresnel term we use the Schlick approximation defined in equation ??, using an refractive index of 1.5 since this is close to the measured value from snake sheds. Our BRDF values are scaled by a shadowing function as described in the appendix of the paper[Sta99]. The reason or this is that the nanostructure of a snake skins is grooved forming V-cavities. therefore some regions on the snake surface is shadowed. Last, we transform our colors from the  $CIE_{XYZ}$  colorspace to the  $CIE_{RGB}$  space using the CIE Standard Illuminant  $D65$ . Last we apply a gamma correction on our computed RGB color values. Consult the section 1.4.3 for further insight.

---

<sup>27</sup>This color weight table contains data for lambda steps in size of 1nm

<sup>28</sup>This extrema values were precomputed in Matlab during the precomputation stage and are passed as an input to our fragment shader.

## 1.4 Technical Details

### 1.4.1 Texture Lookup

In our fragment shader we want to access DFT coefficients of our height field at a given location  $(u, v)$  (defined like in equation ??) to compute structural colors according to equation ??.

For any given nano-scaled surface patch  $P$  with a resolution of  $A$  by  $A \mu\text{m}$ , stored as a  $N$  by  $N$  pixel image  $I$ , one pixel in any direction corresponds to  $dH = \frac{A}{N} \mu\text{m}$ . In Matlab we computed a series of  $n$  output DFT terms  $\{I_{out_1}, \dots, I_{out_n}\}$  from this image  $I$ , where the  $n$ -th image represents the DFT coefficients for the  $n$ -th DFT term in our Taylor Series approximation. Notice, that every DFT image value  $(u, v)$  is in the range  $[-\frac{f_s}{2}, \frac{f_s}{2}]^2$  where  $f_s$  is the sampling frequency equal to  $\frac{1}{dH}$ .

In order to get access to these images within our shader, we have to pass them as GLSL textures. In a GLSL shader the texture coordinates are normalized, which means that the size of the texture maps to the coordinates on the range  $[0, 1]$  in each dimension. By convention the bottom left corner of an image has the coordinates  $(0, 0)$ , whereas the top right corner has the value  $(1, 1)$  assigned.

However,  $u, v$  coordinates are assumed to be in range  $[-\frac{f_s}{2}, \frac{f_s}{2}]$ , but GLSL texture coordinates are in range  $[0, 1]$ . Therefore, for every computed lookup coordinate pair  $(u, v)$ , we have to apply an affine transformation to be in the correct range. Figure 1.10. illustrates this issue of different lookup ranges. In general, an affine transformation is a mapping of the form  $f(x) = sx + b$  where  $s$  is a scaling and  $b$  is a translation.

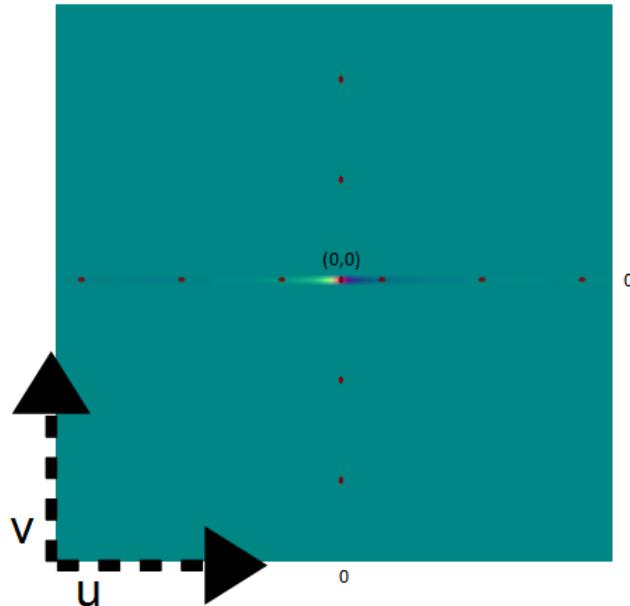


Figure 1.10: Illustration of  $(u, v)$  lookup using GLSL textures.

Figure 1.10 visualized a particular DFT coefficients image for a blazed grating. In GLSL texture coordinates the range of such an image is  $[0, 1]^2$  whereas in a  $(u, v)$  coordinates system this would

correspond to the range  $[-\frac{f_s}{2}, \frac{f_s}{2}]^2$ . Regarding the way the DFT terms were computed in matlab, their zero frequency, i.e.  $(u, v) = (0, 0)$ , is located at the center of the given DFT image. Since by convention the bottom left corner of a GLSL texture corresponds to the value  $(0, 0)$  but we want to work with  $(u, v)$  coordinates, we have to introduce an helper coordinate system  $(u_{lookup}, v_{lookup})$ . This helper coordinates can be computed by a affine transformation applied on  $(u, v)$  coordinates. Hence, we will explain how to compute the scaling  $s$  and translation  $b$  components of our affine transformation:

**Translation component  $b$  of affine transformation:**

Since the zero frequency component of DFT images is shifted towards its centre position, we have to shift the coordinates  $u$  and  $v$  to the center of the current  $N$  by  $N$  pixel image by a bias  $b$ . The translation  $b$  is a constant value and is computed like the following:

$$b = \begin{cases} \frac{N}{2} & \text{if } N \equiv_2 0 \\ \frac{N-1}{2} & \text{otherwise} \end{cases} \quad (1.5)$$

**Scaling component  $s$  of affine transformation:**

For the scaling part of our affine transformation, we have to think a little further: let us consider a  $T$  periodic signal in time, i.e.  $x(t) = x(t + nT)$  for any integer  $n$ . After applying the DFT, we have its discrete spectrum  $X[n]$  with frequency interval  $w_0 = 2\pi/T$  and temporal interval  $dH$ .

Let  $k$  denote the wavenumber which is equal to  $\frac{2\pi}{\lambda}$  for a given wavelength  $\lambda$ . Then the signal is both, periodic with time period  $T$  and discrete with temporal interval  $dH$ . This implies that its spectrum should be discrete with frequency interval  $w_0$  and periodic with frequency period  $\Omega = \frac{2\pi}{dH}$ . This gives us an idea how to discretize the spectrum. For any surface patch  $P$  which is periodically distributed on its surface, its frequency interval along the x-axis is equal to:

$$w_0 = \frac{2\pi}{T} = \frac{2\pi}{N \cdot dH} \quad (1.6)$$

Thus, only wavenumbers that are integer multiples of  $w_0$  after a multiplication with  $u$  must be considered, i.e.  $ku$  is integer multiple of  $w_0$ . Hence the lookup for the  $u$ -direction will look like:

$$\begin{aligned} s(u) &= \frac{ku}{w_0} \\ &= \frac{kuNdH}{2\pi} \\ &= \frac{uNdH}{\lambda} \end{aligned} \quad (1.7)$$

The lookup for the  $v$ -direction will be equal  $s(v)$  defined like in equation 1.7.

**Final affine transformation:**

Using the translation from equation 1.5 and the scaling from equation 1.7, the tranfromed texture lookup-coordinates  $(u_{lookup}, v_{lookup})$  for a given wavelength  $\lambda$  is equal to:

$$\begin{aligned}
(u_{lookup}, v_{lookup}) &= (s(u) + b, s(v) + b) \\
&= \left( \frac{uNdH}{\lambda} + b, \frac{vNdH}{\lambda} + b \right)
\end{aligned} \tag{1.8}$$

Note that for the Windowing Approach, used in algorithm 3, we are visiting a one-pixel-neighborhood around each  $(u, v)$  coordinate pair. For any position  $(i, j)$  of its neighbor-pixels, these local coordinates  $(u_{lookup}^{local_i}, v_{lookup}^{local_j})$  around the origin  $(u_{lookup}, v_{lookup})$  from equation 1.8 are equal to:

$$(u_{lookup}^{local_i}, v_{lookup}^{local_j}) = (i, j) - (u_{lookup}, v_{lookup}) \tag{1.9}$$

### 1.4.2 Texture Blending

So far, we have seen how to render structural colors caused when light is diffracted on a grating. But usually, many objects, such as a snake mesh, also have a texture associated with. Therefore, will have a closer look how to combine colors of a given texture with computed structural colors. For this purpose we will use a so called texture blending approach. This means that, for each pixel, its final rendered color is a weighted average of different color components, such as the diffraction color, the texture color and the diffuse color. In our shader the diffraction color is weighted by a constant  $w_{diffuse}$ , denoting the weight of the diffuse color part. The texture color is once scaled by the absolute value of the Fresnel Term  $F$  and once by  $1 - w_{diffuse}$ . Algorithm 4 shows in depth how our texture blending is implemented:

---

#### Algorithm 4 Texture Blending

---

**Input:**  $c_{texture}$  : Texture color at given fragment position  
 $c_{diffraction}$  : Structural color at given fragment position  
**Output:**  $c_{out}$  : Mixed fragment texture- and structural-color

```

1:  $\alpha = abs(F)$ 
2: if ( $\alpha > 1$ ) then
3:    $\alpha = 1$ 
4: end if
5:  $diffraction_{diff} = (1 - w_{diffuse}) \cdot c_{diffraction}$ 
6:  $text_{spec} = (1 - \alpha) \cdot c_{texture}$ 
7:  $text_{diff} = w_{diffuse} \cdot c_{texture}$ 
8:  $c_{out} = diffraction_{diff} + text_{spec} + text_{diff}$ 

```

---

### 1.4.3 Color Transformation

In our fragment shader, we access a table which contains precomputed CIE's color matching functions values<sup>29</sup> from  $\lambda_{min} = 380nm$  to  $\lambda_{max} = 780nm$  in  $5nm$  steps. In algorithm 3 we describe how to compute the  $CIE_{XYZ}$  color values as described in section ???. We can transform the color values into  $CIE_{sRGB}$  gammut by performing the following linear transformation:

---

<sup>29</sup>Such a function value table can be found at [cvrl.ioo.ucl.ac.uk](http://cvrl.ioo.ucl.ac.uk) for example

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = M \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \quad (1.10)$$

where one possible transformation is:

$$M = \begin{bmatrix} 0.41847 & -0.15866 & -0.082835 \\ -0.091169 & 0.25243 & 0.015708 \\ 0.00092090 & -0.0025498 & 0.17860 \end{bmatrix} \quad (1.11)$$

One aspect we have to keep in mind is what is the tristimulus value of the color defining our white point in our images. Defining what tristimulus value whit corresponds to, usually depends on the application. For our shaders we use use the CIE Standard Illuminant  $D65$ .  $D65$  is intended to represent an average midday sun daylight. Applying the  $D65$  illuminant, the whole colorspace transformation will look like:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = M \cdot \begin{bmatrix} X/D65.x \\ Y/D65.y \\ Z/D65.z \end{bmatrix} \quad (1.12)$$

Each component of the Standard Illuminant acts as rescaling factor according to the whitepoint definiton of  $D65$  for our computed color values. Last, we perfrom a gamma correction on each pixel's  $(R, G, B)$  value. A gamma correction is a non linear transformation which controls the overall brightness of an image<sup>30</sup>.

## 1.5 Discussion

### 1.5.1 Comparison: per Fragment-vs. per Vertex-Shading

In this chapter we have seen how our render is implemented. We discussed our fragment shader which is responsible for computing the structural color values resulting by diffracted light. Our fragment shader's runtime performance depends on the resoultion of the final rendered image, since there is a correspondance between pixels and fragments. Notice that we also could have computed the strucural colors during the vertex shading process. Then, the whole rendering performance of our shading approach would only depend on the vertex count of our meshes. Shading on a per vertex basis is usually bad since in order to get a nice and accurate rendering, the vertex count of a mesh has to be big. Furthermore, a vertex shader produces poor results for shapes like a cube for example, according to our structural color model. Therfore, shading on a per fragment shader scales depending on the rendered image resolution and is independent of the mesh vertices (their distribution and count).

### 1.5.2 Optimization of Fragment Shading: NMM Approach

The fragment shader algorithm described in algorithm 3 performs a gaussian window approach by sampling over the whole wavelength spectrum in uniform step sizes. This algorithm slow, since we for each pixel we iterate over the whole lambda spectrum. Furthermore, for any pixel, we iterate over its one-neighborhood. When also considering the number of taylor series approximation steps, we will have a run-time complexity of

---

<sup>30</sup>For further information about gamma correction, please have a look in the book *Fundamentals of Computer Graphics*[PS09].

$$O(\#spectrtumIter \cdot \#taylorIter \cdot neighborhoodRadius^2) \quad (1.13)$$

Our goal is to optimize this runtime behaviour. Instead sampling over the whole wavelength spectrum, we instead could integrate over a minimal number of wavelengths contributing the most to our shading result like shown in figure 1.11. These values are elicited like the following: Lets consider  $(u, v, w)$  defined as in equation ???. Let  $d$  be the spacing between two slits of a grating. For any  $L(\lambda) \neq 0$  it follows  $\lambda_n^u = \frac{du}{n}$  and  $\lambda_n^v = \frac{dv}{n}$ . Therefore we can derive the following boundaries of  $n$ :

$$\begin{aligned} \text{If } u, v > 0 \quad N_{min}^u &= \frac{du}{\lambda_{max}} \leq n_u \leq \frac{du}{\lambda_{min}} = N_{max}^u \\ N_{min}^v &= \frac{dv}{\lambda_{max}} \leq n_v \leq \frac{dv}{\lambda_{min}} = N_{max}^v \end{aligned}$$

$$\begin{aligned} \text{If } u, v < 0 \quad N_{min}^u &= \frac{du}{\lambda_{min}} \leq n_u \leq \frac{du}{\lambda_{max}} = N_{max}^u \\ N_{min}^v &= \frac{dv}{\lambda_{min}} \leq n_v \leq \frac{dv}{\lambda_{max}} = N_{max}^v \end{aligned}$$

$$\begin{aligned} \text{If } (u, v) = (0, 0) \quad n_u &= 0 \\ n_v &= 0 \end{aligned}$$

By transforming those equations in the equation collection ?? to  $(\lambda_{min}^u, \lambda_{max}^u)$  and  $(\lambda_{min}^v, \lambda_{max}^v)$  respectively, we can reduce the total number of required iterations in our fragment shader. We denote this optimization by the  $n_{min}, n_{max}$  (NMM) shading approach.

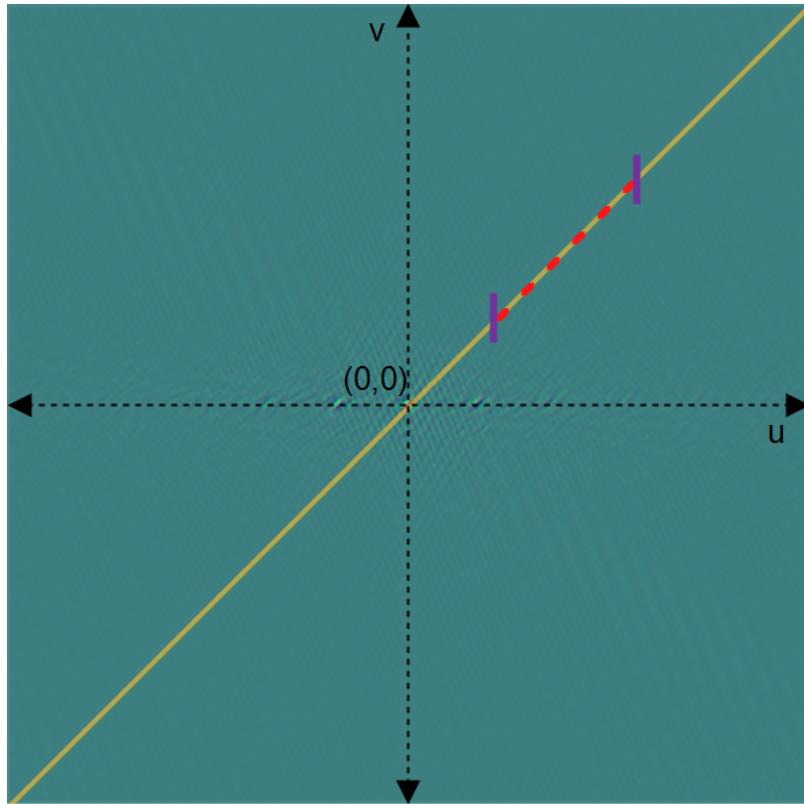


Figure 1.11: Illustration of NMM shading approach's idea.

Figure 1.11 illustrated the idea of the NMM shading approach by showing a visualization of a DFT term. Instead integrating over the whole wavelength spectrum (yellow line), we can find a constrained wavelength range by using equation ??, and then only integrate over a reduced, discrete set of  $\lambda$  values (indicated by the dotted red line). Since we are uniformly integrating over the wavelength spectrum but this approach selects non-uniformly a reduced wavelength spectrum, this can lead to issues. Close to zero frequency, i.e.  $(u, v)$  equal to  $(0, 0)$ , we could have too few or even no samples, using the NMM approach. Thus, our workaround according to this issue is to use a default color close to the zero frequency. Technically, this means that return white color around a small  $\epsilon$  circumstance.

### 1.5.3 The PQ Shading Approach

Another variant is the *PQ* approach described in section ???. In section ?? we described how to interpolate the values relying on sinc functions. Another approach would be simple to perform a linear interpolation. This last variant does not require to iterate over a pixel's neighborhood. This it has a faster runtime complexity faster than any windowing approach. Using the sinc function interpolation is well understood in the field of signal processing and will give us reliable results. The drawback of this approach is that we again have to iterate over a pixel's neighborhood within the fragment shader. This once again will slow down the whole shading. Algorithm 5 describes the modification of the fragment shader in algorithm 3 in oder to use sinc interpolation for the *PQ* approach instead using a gaussian window:

---

**Algorithm 5** Sinc interpolation for PQ approach
 

---

**Input:** same input as in algorithm 3  
**Output:**  $c_p$  : Structural Color based in the PQ approach (See section ??)

```

1: Foreach Pixel  $p \in Image I$  do
2:    $w_p = \sum_{(i,j) \in \mathcal{N}_1(p)} sinc(\Delta_{p,(i,j)} \cdot \pi + \epsilon) \cdot I(i,j)$ 
3:    $c_p = w_p \cdot (p^2 + q^2)^{\frac{1}{2}}$ 
4: end for
```

---

In a fragment shader, for every fragment  $p$  we compute its reconstructed function value  $f(p)$  stored in  $w_p$ . The value  $w_p$  is the reconstructed signal value at  $f(p)$  by the sinc function as described in section ???. Similar like for the gaussian windowing approach, we calculate the distance  $\Delta_{p,(i,j)}$  between the current fragment position  $p$  and each of its neighbors  $(i,j) \in \mathcal{N}_1(p)$  in its one-neighborhood. Multiplying this distance by  $\pi$  gives us the an angle used for the sinc function interpolation. Notice that, in order to avoid division by zeros side-effects, we add a small integer  $\epsilon$  to our angle value.

# Chapter 2

## Results

In this chapter we examine the rendered output results of our implementation of our BRDF models applied to different input patches such as Blaze grating or Elaphe ?? and Xenopeltis ?? snake nano-scaled surface sheds. We are discussing and comparing both, their BRDF maps 2.1 and the corresponding renderings on a snake geometry like shown in section 2.2 for various input parameters. Last we also show a real experimental image showing the effect of diffraction for similar parameters like we have.

### 2.1 BRDF maps

A BRDF map shows a shader's output for all possible viewing directions for a given, fixed, incident light direction. We assume that each viewing direction is expressed in spherical coordinates (See appendix ??)  $(\theta_v, \phi_v)$  and is represented in the map at point

$$(x, y) = (\sin(\theta_v)\cos(\phi_v), \sin(\theta_v)\sin(\phi_v)) \quad (2.1)$$

with its origin at the map-center. The light direction for normal incidence  $(\theta_i, \phi_i)$  has been fixed to  $(0, 0)$  for our rendered results.

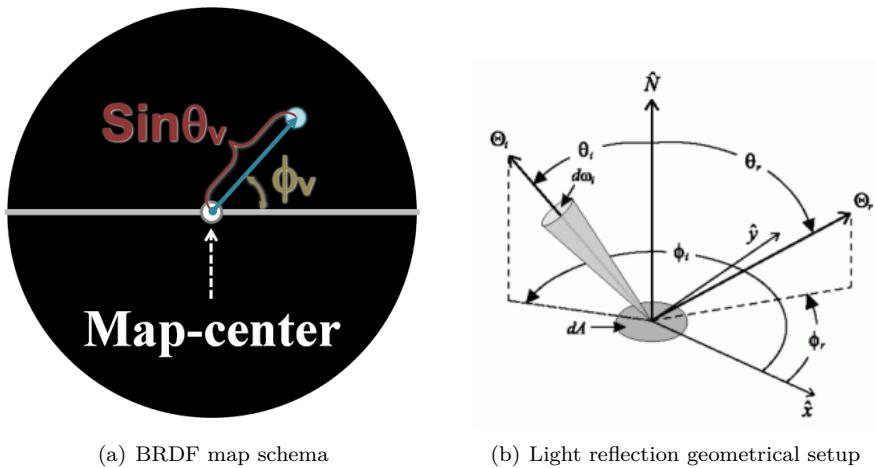


Figure 2.1: BRDF maps<sup>1</sup> for different patches:  $\Theta = (\theta_i, \phi_i)$  is the direction of light propagation

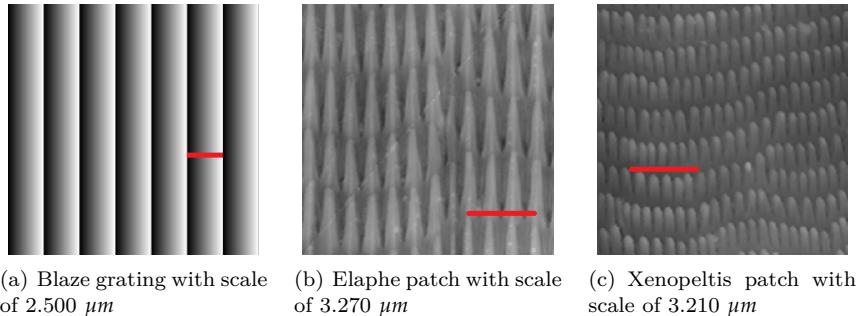


Figure 2.2: Cutouts of our nano-scaled surface gratings used for rendering within our shader with a scale indicator (red line) for each patch. Note that for rendering, we use larger patches.

Figure 2.3 shows the BRDF maps of the full lambda space sampling approach (**FLSS** like introduced in section 1.3.2) as described in section 1.3.2 applied on different nanoscale surface gratings as shown in figure 2.2. In Subfigure 2.3(a) we see the BRDF map for the Blazed grating, showing high relative brightness for its first order diffraction, i.e. for the Blazed gratings most of the diffracted spectral energy lies in its first order. Notice that the surface of blazed grating is forming a step structure for which the angle between the step normal and the grating normal is denoted by *blaze angle*. Every blazed grating is manufactured in the Littrow<sup>2</sup> configuration. This means that the blaze angle is chosen such that the diffraction angle and incidence angle are identical. Thus it a blazed grating has maximize efficiency for the wavelength of the used light. Higher diffraction modes are still perceivable (second and higher diffraction orders) but with a much lower relative

<sup>1</sup>image source of figure:

- 2.1(a): Taken from D.S.Dhillon's Paper [DSD14b]
  - 2.1(b): Taken from <http://math.nist.gov/~FHunt/appearance/brdf.html>

<sup>2</sup>For further information please see [http://en.wikipedia.org/wiki/Blazed\\_grating](http://en.wikipedia.org/wiki/Blazed_grating).

brightness. The asymmetry of the pattern is due to the asymmetric geometry of the grating 2.2(a).

The finger-like structures contained in the Elaphe surface grating 2.2(b) are quite regularly aligned and hence diffraction occurs along the horizontal axis for the BRDF map as shown in figure 2.3(b). The reason for not seeing any strong diffraction color contribution along other directions in the BRDF map is due to the fact that these ‘nano-fingers’ overlap across layers and thus do not exhibit any well-formed periodicity along finger direction.

For Xenopeltis surface grating 2.2(c), we observe diffraction along many different, almost vertical directions in the BRDF map 2.3(c) since the layers of the finger-like structures do not overlap and are shifted significantly along their length but still exhibit some local consistency. A similar argument holds true for diffraction across locally periodic finger patches with slightly different orientations.

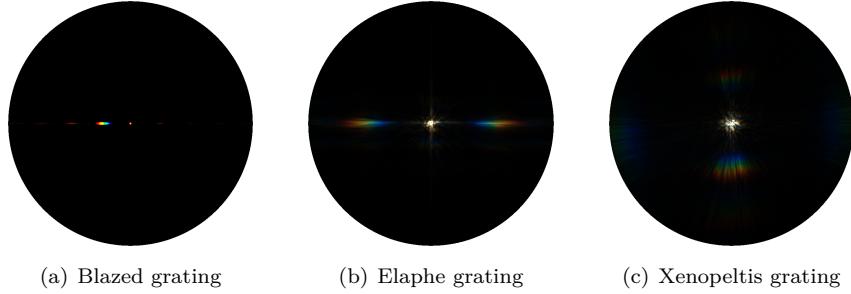


Figure 2.3: BRDF maps for different patches

Figure 2.4 shows BRDF maps of all our BRDF models applied on the Blaze grating. Figure 2.4(a) shows the FLSS shading approach result for our blazed grating and it is used in order to compare with our other rendering approaches.

Figure 2.4(b) shows the BRDF map for the NMM approach, introduced in section 1.5.2, which is close to the FLSS approach, verified in section ??, just like in the case of corresponding evaluation in figure ???. Nevertheless there is a small, noticeable difference:

since in the actual technical implementation of the NMM shading approach treats the center of the BRDF map as a separate case, i.e. everything around a small  $\epsilon$ -circumference has white color assigned we note a white circular spot around the map center. Except this fact, the BRDF map resembles the FLSS results.

Figure 2.4(c) shows the BRDF map for the PQ approach which relies on sinc-interpolation. The PQ BRDF map and the FLSS results are visual alike. Compared to the evaluation plots in figure ???, the BRDF maps even persuade more. Compared to FLSS, one difference we notice is that the first order of diffraction is a little spread for the PQ approach. Without<sup>3</sup> applying a sinc-interpolation, this spreading effect would be even strengthened.

Last, let us consider figure 2.4(d) which shows the BRDF map produced by using Nvidia Gem’s implementation [JG04] of Stam’s BRDF model when constraining the y-axis of the BRDF map.

<sup>3</sup>Note that if we do not perform a sinc-interpolation this would correspond to apply a linear interpolation instead.

This corresponds to a 1d diffraction grating, along the x-axis. This model only uses the spacing  $d$  of a given grating. It also always produces highly symmetric results.

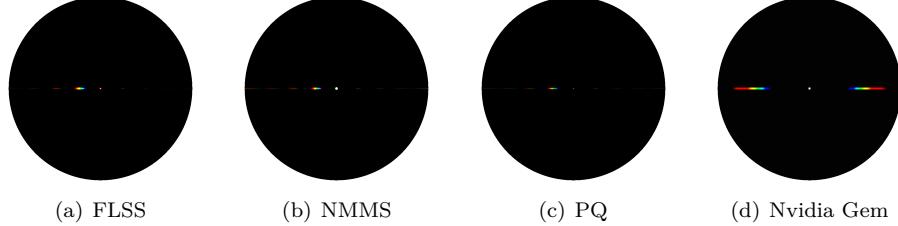


Figure 2.4: BRDF maps for Blazed grating comparing our different rendering approaches

Figure 2.5 and figure 2.6 show the BRDF maps for different wavelength step sizes used in the fragment shader for the FLSS approach applied on the blazed grating and the Elaphe snake shed, respectively. Within our fragment shaders the outermost loop iterates over the range [380nm, 780nm] for a given step size  $\lambda_{step}$  to integrate over the wavelength spectrum. Having bigger step sizes implies having fewer  $\lambda$ -samples which will reduce the overall runtime of a shader but, it will also introduce artifacts and therefore lower the overall shading quality. For Elaphe surface grating, artifacts are perceivable when  $\lambda_{step} \leq 10\text{nm}$ . Results produced by using 5nm step sizes do not differ from those produced by using  $\lambda_{step} = 1\text{nm}$ . This allows us to set  $\lambda_{step}$  at 5nm. For a Blazed grating we may choose even bigger step sizes without losing any rendering quality (see figure 2.5).

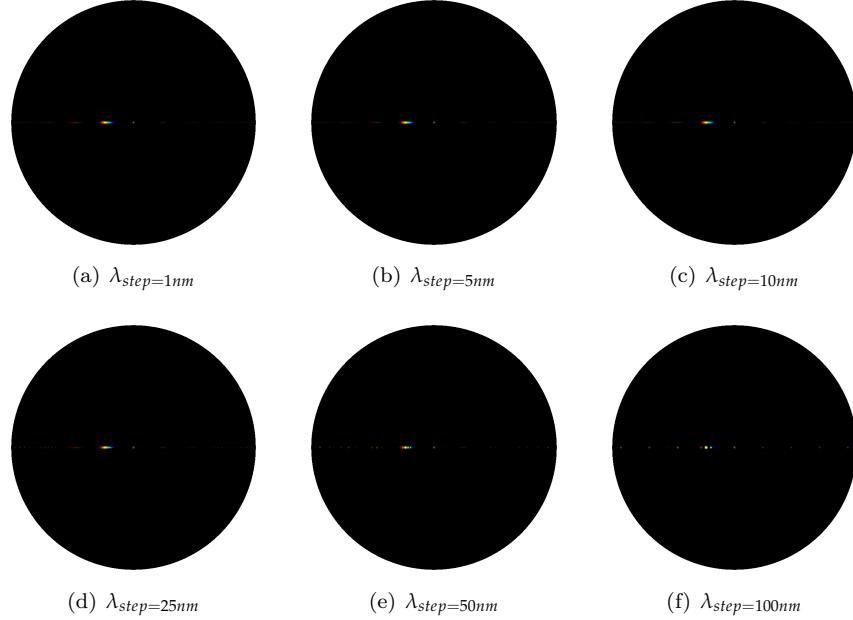
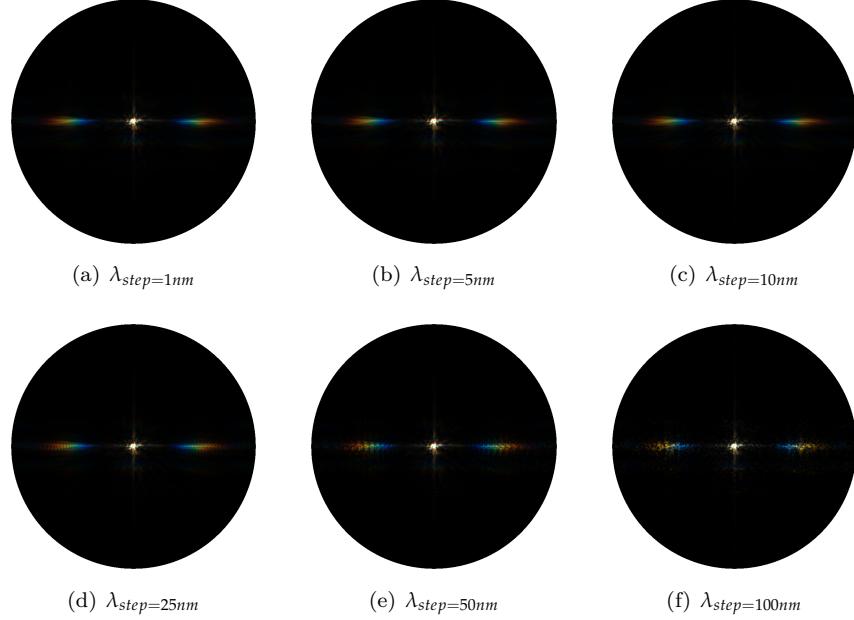


Figure 2.5: Blazed grating at  $2.5\mu\text{m}$ : Different  $\lambda$  step sizes

Figure 2.6: Elaphe grating at  $65\mu m$ : Different  $\lambda$  step sizes

The Figures 2.7, 2.8, 2.9 show a comparison of the BRDF maps produced by the FLSS approach (on the left) and the PQ shading approach (on the right) applied on all our patches. For Blazed grating, as already mentioned, we notice that both approaches, FLSS and PQ, resemble each other. We also notice that for PQ map, the first order diffraction color contribution is spread. For Elaphe and Xenopeltis grating we notice similar shaped BRDF patterns, even when the angle of light varies, but nevertheless, they also contain some artifacts. For the PQ approach, this also holds true when we do not apply we a sinc-interpolation like shown in figure 2.7(c).

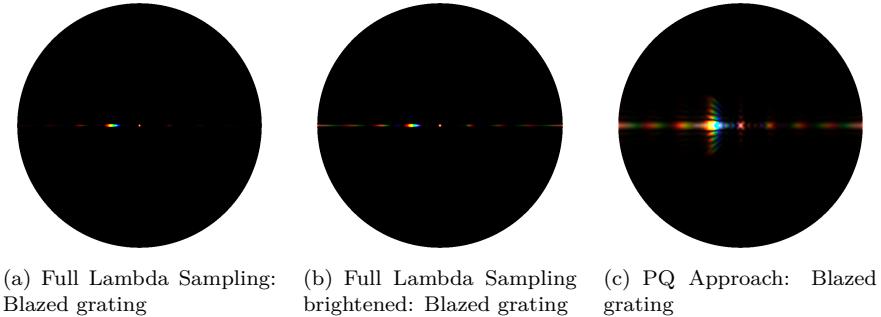


Figure 2.7: Blazed grating: PQ approach vs full lambda space sampling

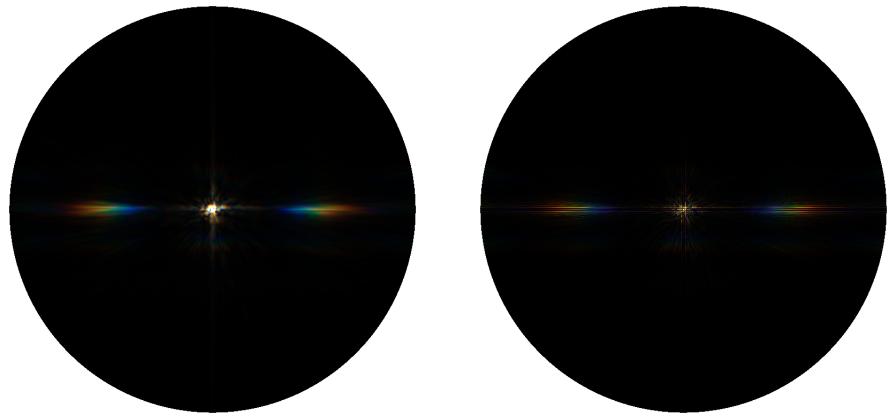


Figure 2.8: Elaphe grating: PQ approach vs full lambda space sampling

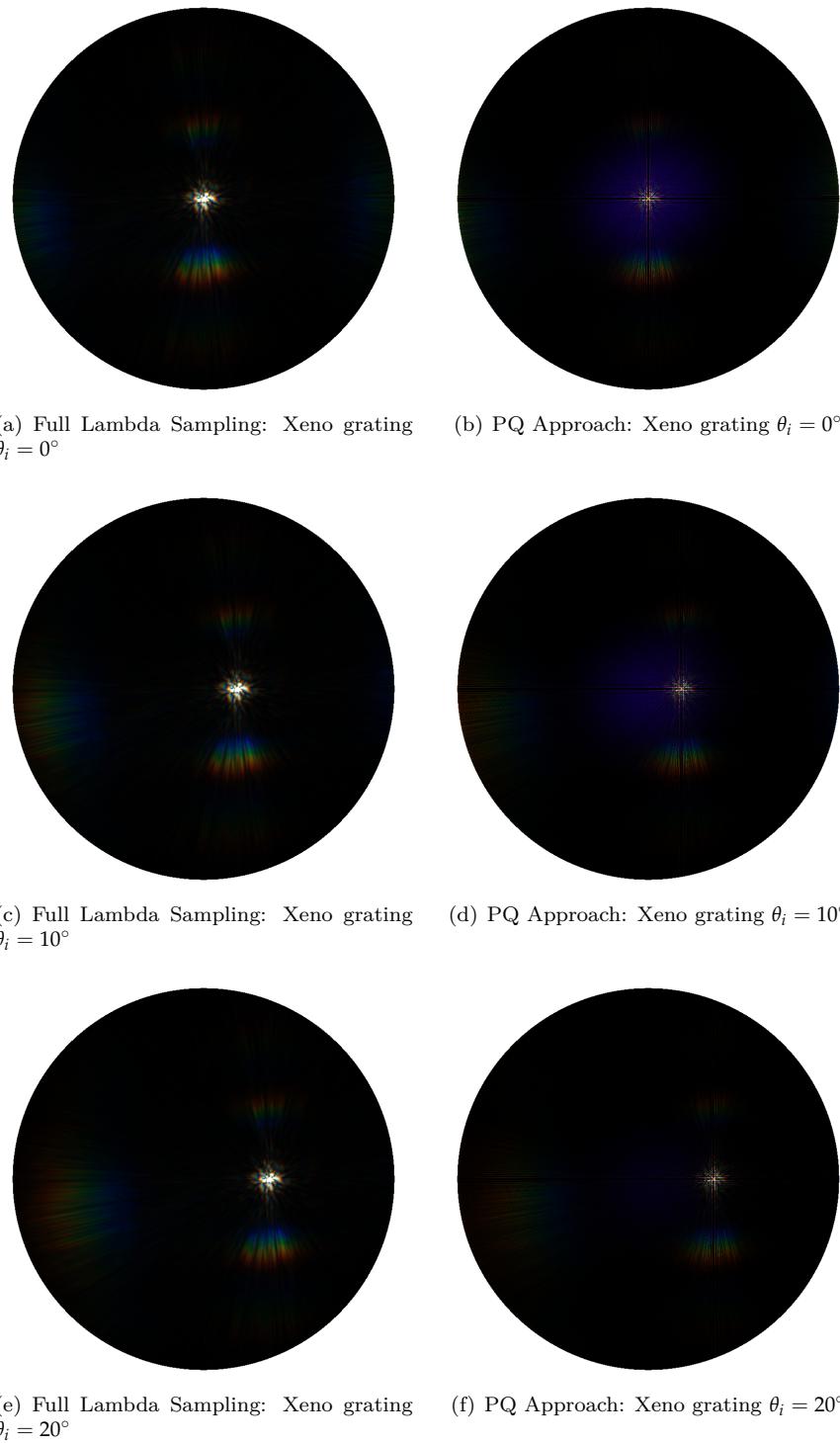


Figure 2.9: Xeno grating: PQ approach vs full lambda space sampling

Figure 2.10 shows BRDF maps for the full lambda sampling approach applied to the Blazed grating, while varying the value for the spatial variance  $\sigma_s$ . This akin to changing the coherence length for the incident light. The lower the coherence length, the fewer interacting grating periods produce blurred diffraction bands for different  $\lambda$  which overlap to produce poorly resolved colors.

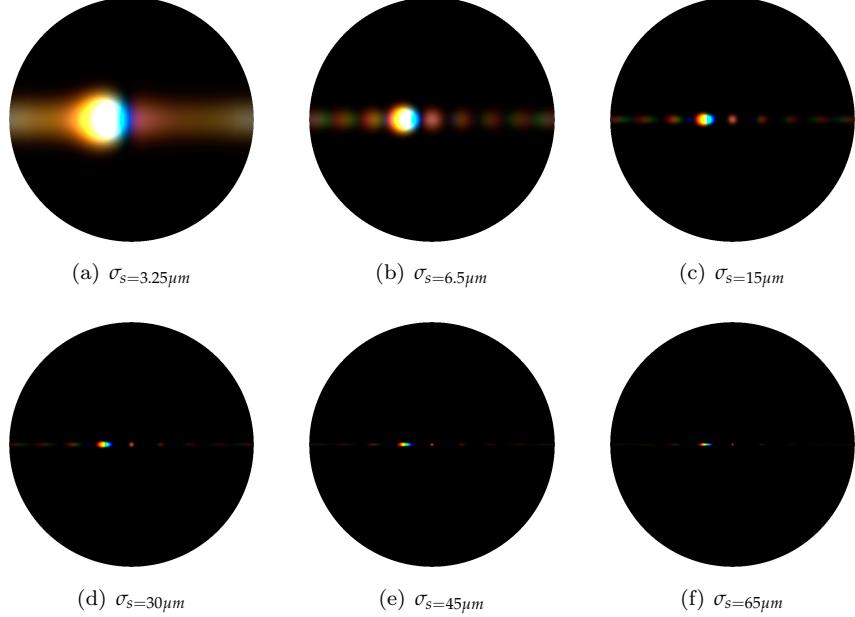


Figure 2.10: Blazed grating with periodicity of  $2.5\mu m$ : Different  $\sigma_s$

Figures 2.11 and 2.12 show the BRDF maps the reference-, FLSS approach using different values for  $N$  in the taylor series approximation. For both input patches we clearly visually observe the convergence of the taylor series for higher values of  $N$ . We visually observe convergence of the Taylor series for all our patches a very large value of  $N^4$ . In section ?? we already provided a proof for the convergence for a specific geometrical set up which could be applied easily to another setup. The higher the value of  $N$  the less the BRDF map changes. For the Blazed grating, we do not see much changing patterns for  $N \geq 7$  and for the Elaphe surface grating  $N \geq 9$ . In general it is rather hard to say for what value of  $N$  we have visual convergence since in algorithm 1 we compute the inverse FFT of  $power(1j * patchImg, t)$  which is equivalent to  $iFFT(patchImg)^n \cdot i^n$ . Since we multiply this by  $i^n$  there are four possible series where each converges within its own convergence radius.

---

<sup>4</sup>Using N equal to 39 lead to visual convergence for all our used gratings.

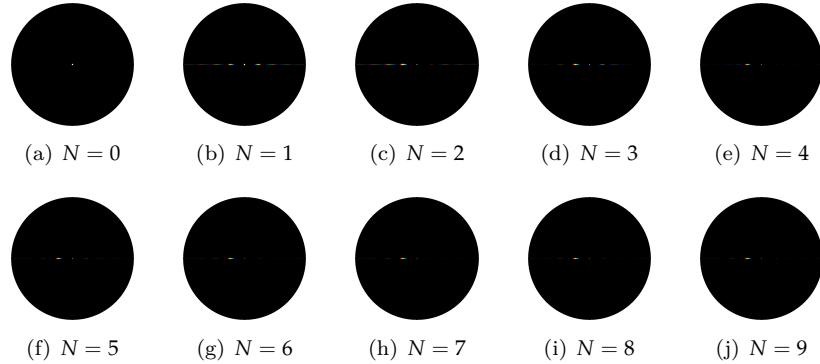
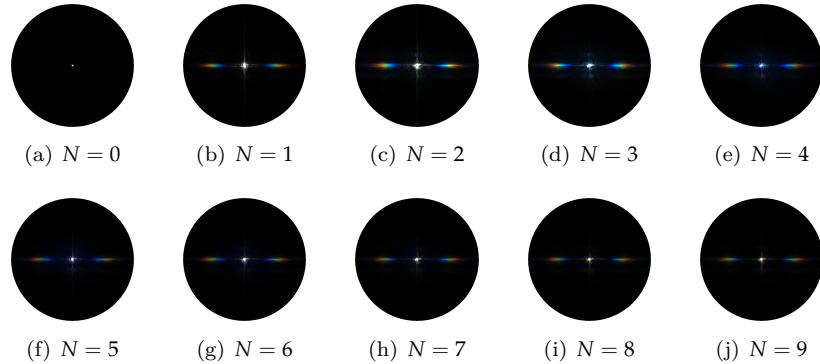
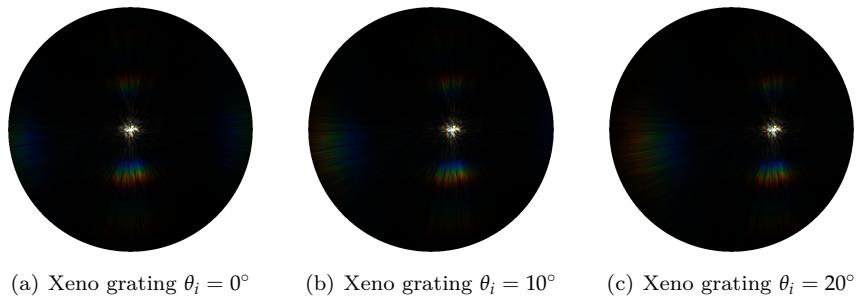
Figure 2.11: Blazed grating at  $2.5\mu\text{m}$ :  $N$  Taylor IterationsFigure 2.12: Elaphe grating at  $65\mu\text{m}$ :  $N$  Taylor Iterations

Figure 2.13 shows the BRDF maps of the full lambda shading approach applied on the Xenopeltis snake shed, using different  $\theta_i$  incident angles. When slightly moving the incident angle  $\theta_i$ , we can observe how the brdf map changes. For higher values of  $\theta_i$  we start seeing diffraction color contribution on the right side of the BRDF map.

Figure 2.13: BRDF maps for Xeno grating: different  $\theta_i$  angles

## 2.2 Snake surface geometries

In this section we are going to present our actual renderings simulating the effect of diffraction caused when a directional light source encounters different nano-scaled surfaces on a given curved snake mesh. We will see that diffraction colors change dramatically with changes in light direction, surface normals and viewing direction, which is typical for diffraction colors observed in nature. For rendering we are going to rely on our full wavelength space sampling approach. Unfortunately, this approach is rather slow and can barely be considered as being interactive performing. Nevertheless, we have introduced some optimizations in order to become interactive in rendering, such as the  $N_{min}, N_{max}$  approach, we are going to use this slow approach since this resembles the ground truth and therefore is the most accurate among all our presented approaches. As mentioned we are going to render diffraction on a given snake mesh. Note that we actually just have one particular mesh, for all our renderings we are going to use the same snake mesh which has been produced by 3d scanning an Elaphe snake species, consisting of 11696 vertices and 22950 faces. The reason for that is that it was hard to get a Xenopeltis snake ready for being 3d scanned. In addition, the micro-geometry is highly similar among snake species, it is the geometry of the nano-structures that are highly different among species and that cause the snake to be or not be iridescent. So, even Xenopeltis would not give you very different geometry than Elaphe. Table 2.1 lists the system specifications of the machine I used in order to produce the rendered images.

Processor	Intel i7 CPU 970 @ 3.20 GHz (12 CPUs)
Memory	12288 MB RAM
Graphics Card	GeForce GTX 770

Table 2.1: Hardware specifications of the machine which produced rendered results. Statistics are provided using the tool NVIDIA Geforce Experience.

Figure 2.14 shows renderings produced by the full lambda sampling approach applied on a snake shaped mesh for different given input patches. Due to the fact that a Blazed grating has its maximum intensity for a certain direction and the geometry of the snake mesh is curved which means non-flat, we can expect rather less diffraction color contribution like shown in figure 2.14(b). Differently for our other two gratings, Elaphe and Xenopeltis. For both renderings, we can see color contribution despite the effect of diffraction whereas we see much less colorful patterns for Elaphe 2.14(b) than for Xenopeltis 2.14(c). This also corresponds to the reality, considering the figure ?? as a reference. The nano-scaled surface structure of the species Elaphe as shown in figure 2.2(b) does not look that regular under the electron scanning microscope. This is why it is much less iridescent than the other species. Xeno has a brownish body with no pattern that makes the iridescence more spectacular than on Ellaphe.

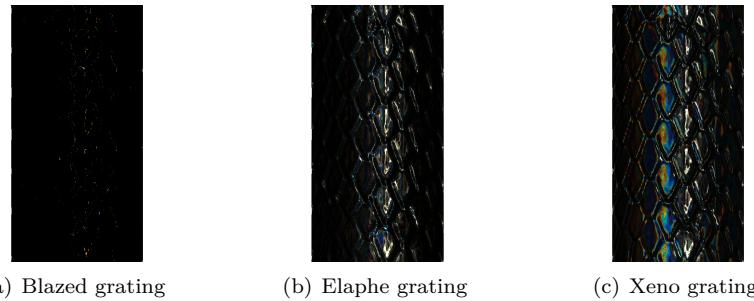


Figure 2.14: Diffraction of different snake skin gratings rendered on a snake geometry

Figure 2.15 contains a summary-collection of subfigures for rendering the effect of diffraction produced by the full lambda shading approach with all its required components, applied on our snake mesh, using the Elaphe nano-scaled surface structure. Subfigure 2.15(b) shows the final diffraction color-contribution result with texture-blending. We only see little diffraction color contribution in this subfigure which resembles quite well to the reality as shown in figure ???. In subfigure 2.15(d) we see the light cone in order to show the direction of the light source besides the rendered results. Subfigure 2.15(e) is a sample Fourier image of Elpahe's nanosclae surface structure 2.15(d).

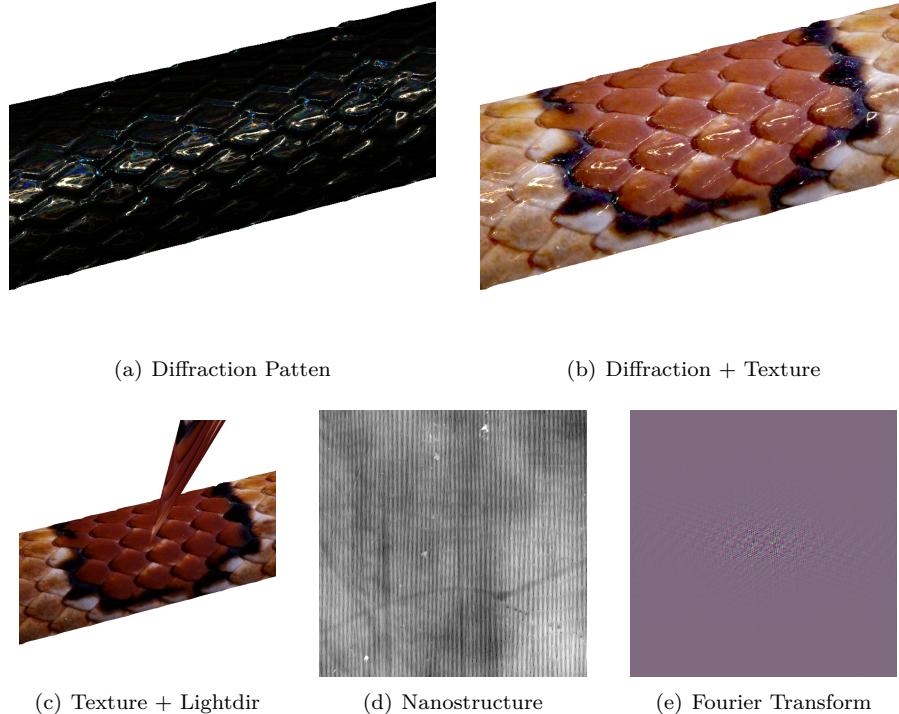


Figure 2.15: Diffraction for Elaphe snake skin

Like in the previous figure this figure 2.16 also shows a summary-collection of subfigures for the effect of diffraction with all its involved components but this time for the Xenopeltis snake surface. For texture blending we use the same texture like we used for Elaphe. For Xenopeltis see quite a lot color contribution due the phenomenon of diffraction like shown in figure 2.16(b). Comparing this to a real image ?? we notice much resemblance regarding the reflectance strength and colorful pattern.

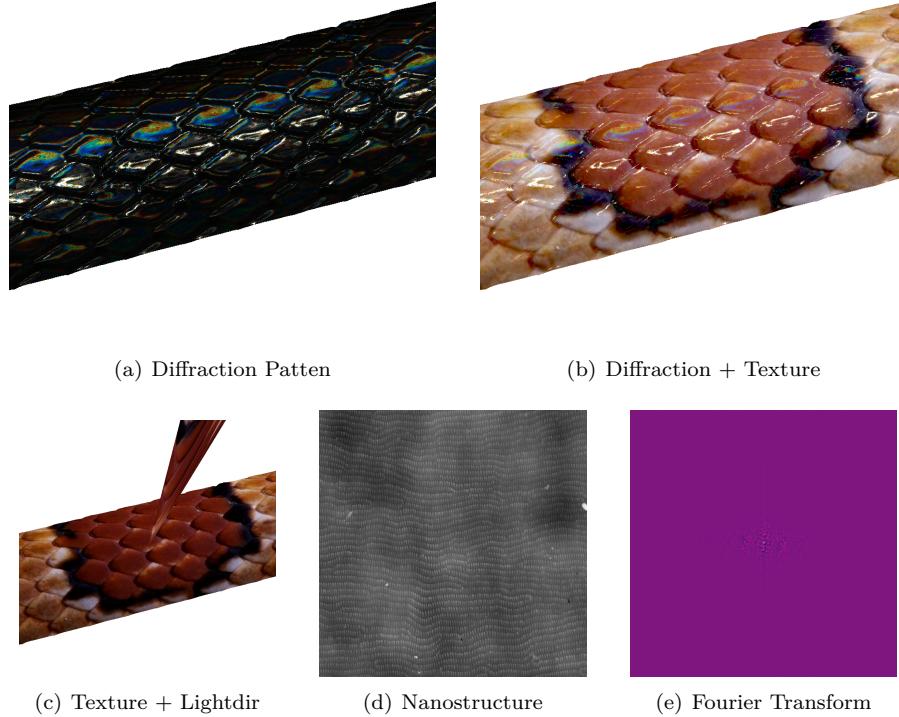


Figure 2.16: Diffraction for Xeno snake skin

Figure 2.17 shows the diffraction pattern of a Elaphe snake shed for different zoom levels for fixed incident light and viewing direction using the full lambda sampling approach. From those different close up perspectives it would appear the complexity of the colorful diffraction pattern.

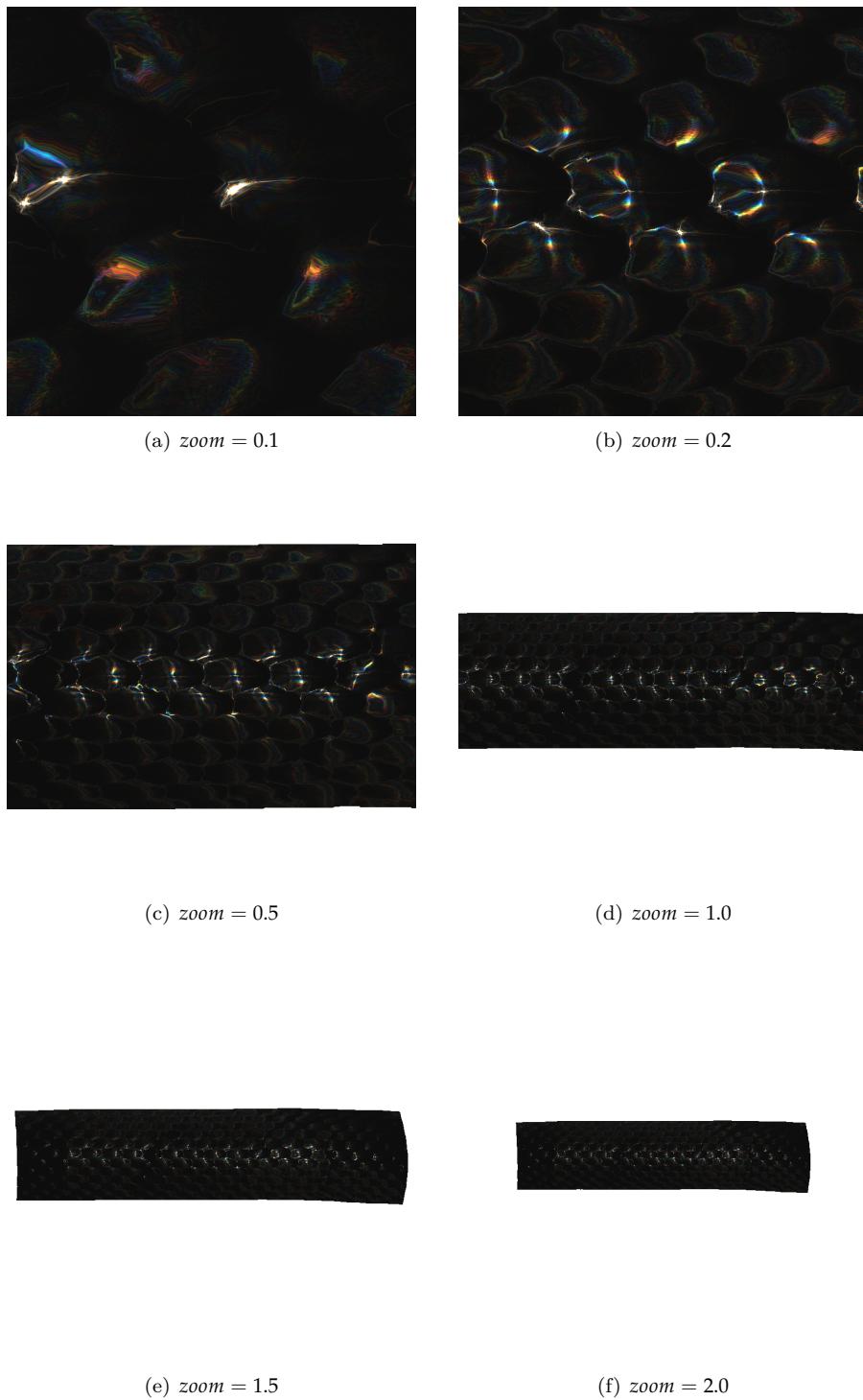


Figure 2.17: Diffraction on Elaphe snake skin grating: Different camera zoom levels

Figure 2.18 shows how the diffraction pattern changes when slightly moving the incident light direction. Which gives us an impression what kind of complex, perspective-dependent pattern the phenomenon of diffraction may cause.

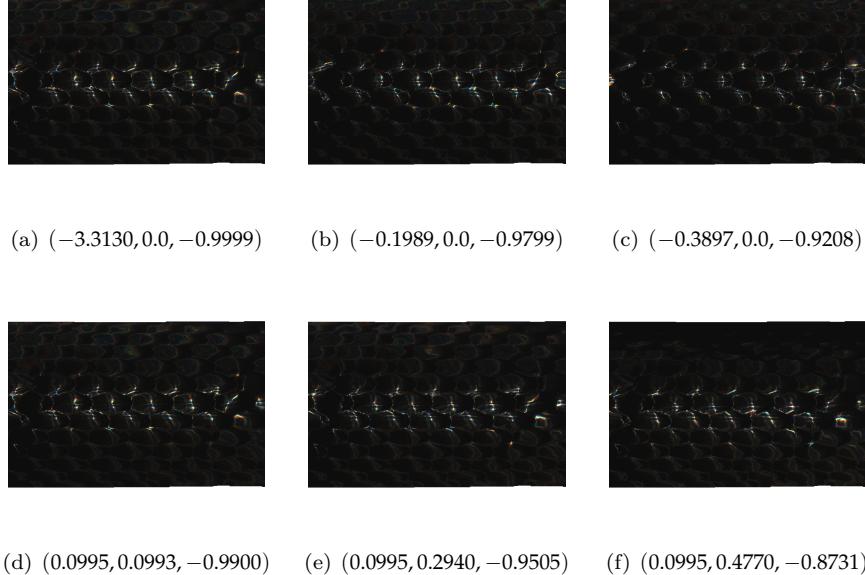


Figure 2.18: Diffraction on Elaphe snake skin grating: Different light directions

Figure 2.19 shows a photo of an experimental setup for demonstrating the effect of diffraction using a Elaphe snake grating. The exact parameters for the experimental setup are unknown. Nevertheless this image gives us an impression of how close our model is to the reality comparing it with our simulated results since we notice similar diffraction patterns for our simulated results using an Elaphe snake shed.

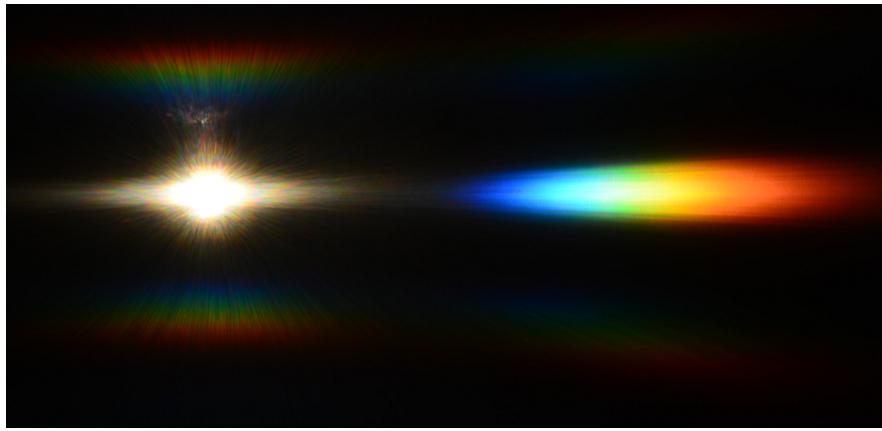


Figure 2.19: Diffraction Elaphe: experimental setup

# List of Tables

2.1 Hardware Specifications . . . . .	33
---------------------------------------	----

# List of Figures

1.1	DFT Terms for a Blazed grating . . . . .	5
1.2	Renderer Architecture . . . . .	6
1.3	Triangular Mesh . . . . .	7
1.4	Vertex Shader . . . . .	8
1.5	Camera Coordinate System . . . . .	9
1.6	Camera Matrix . . . . .	9
1.7	Rays of a Directional Light . . . . .	11
1.8	Triangle Rasterization . . . . .	12
1.9	Fragment Shader . . . . .	13
1.10	Lookup DFT Coefficients in Textures . . . . .	17
1.11	NMM Shading Approach . . . . .	22
2.1	BRDF Map . . . . .	25
2.2	Our Gratings . . . . .	25
2.3	BRDF Map FLSS of our Gratings . . . . .	26
2.4	BRDF Map using our Approaches for Blazed Grating . . . . .	27
2.5	BRDF Map varying step sizes FLSS Blazed Grating . . . . .	27
2.6	BRDF Map varying step sizes FLSS Elaphe Grating . . . . .	28
2.7	Blazed grating: PQ approach vs full lambda space sampling . . . . .	28
2.8	Elaphe grating: PQ approach vs full lambda space sampling . . . . .	29
2.9	Xeno grating: PQ approach vs full lambda space sampling . . . . .	30
2.10	Blazed grating with periodicity of $2.5\mu m$ : Different $\sigma_s$ . . . . .	31
2.11	Blazed grating at $2.5\mu m$ : $N$ Taylor Iterations . . . . .	32
2.12	Elaphe grating at $65\mu m$ : $N$ Taylor Iterations . . . . .	32
2.13	BRDF maps for Xeno grating: different $\theta_i$ angles . . . . .	32
2.14	Diffraction of different snake skin gratings rendered on a snake geometry . . . . .	34
2.15	Diffraction for Elaphe snake skin . . . . .	34
2.16	Diffraction for Xeno snake skin . . . . .	35
2.17	Diffraction on Elaphe snake skin grating: Different camera zoom levels . . . . .	36
2.18	Diffraction on Elaphe snake skin grating: Different light directions . . . . .	37
2.19	Diffraction Elaphe: experimental setup . . . . .	37

# List of Algorithms

1	Precomputation: Pseudo code to generate Fourier terms . . . . .	4
2	Vertex diffraction shader pseudo code . . . . .	10
3	Fragment diffraction shader pseudo code . . . . .	15
4	Texture Blending . . . . .	19
5	Sinc interpolation for PQ approach . . . . .	23

# Bibliography

- [Bar07] BARTSCH, Hans-Jochen: *Taschenbuch Mathematischer Formeln*. 21th edition. HASNER, 2007. – ISBN 978-3-8348-1232-2
- [CT12] CUYPERS T., et a.: Reflectance Model for Diffraction. In: *ACM Trans. Graph.* 31, 5 (2012), September
- [DSD14a] D. S. DHILLON, et a.: Interactive Diffraction from Biological Nanostructures. In: *EUROGRAPHICS 2014 / M. Paulin and C. Dachsbacher* (2014), January
- [DSD14b] D. S. DHILLON, M. Single I. Gaponenko M. C. Milinkovitch M. Z. J. Teyssier T. J. Teyssier: Interactive Diffraction from Biological Nanostructures. In: *Submitted at Computer Graphics Forum* (2014)
- [For11] FORSTER, Otto: *Analysis 3*. 6th edition. VIEWEG+TEUBNER, 2011. – ISBN 978-3-8348-1232-2
- [I.N14] I. NEWTON: *Opticks, reprinted*. CreateSpace Independent Publishing Platform, 2014. – ISBN 978-1499151312
- [JG04] JUAN GUARDADO, NVIDIA: Simulating Diffraction. In: *GPU Gems* (2004). <https://developer.nvidia.com/content/gpu-gems-chapter-8-simulating-diffraction>
- [LM95] LEONARD MANDEL, Emil W.: *Optical Coherence and Quantum Optics*. Cambridge University Press, 1995. – ISBN 978-0521417112
- [MT10] MATIN T.R., et a.: Correlating Nanostructures with Function: Structurnal Colors on the Wings of a Malaysian Bee. (2010), August
- [PAT09] PAUL A. TIPLER, Gene M.: *Physik für Wissenschaftler und Ingenieure*. 6th edition. Spektrum Verlag, 2009. – ISBN 978-3-8274-1945-3
- [PS09] P. SHIRLEY, S. M.: *Fundamentals of Computer Graphics*. 3rd edition. A K Peters, Ltd, 2009. – ISBN 978-1-56881-469-8
- [R.H12] R. HOOKE: *Micrographia, reprinted*. CreateSpace Independent Publishing Platform, 2012. – ISBN 978-1470079031
- [RW11] R. WRIGHT, et a.: *OpenGL SuperBible*. 5th edition. Addison-Wesley, 2011. – ISBN 978-0-32-171261-5
- [Sta99] STAM, J.: Diffraction Shaders. In: *SIGGRPAH 99 Conference Proceedings* (1999), August
- [T.Y07] T. YOUNG: *A course of lectures on natural philosophy and the mechanical arts Volume 1 and 2*. Johnson, 1807, 1807

## Erklärung

gemäss Art. 28 Abs. 2 RSL 05

Name/Vorname: .....

Matrikelnummer: .....

Studiengang: .....

Bachelor

Master

Dissertation

Titel der Arbeit: .....

.....

.....

LeiterIn der Arbeit: .....

.....

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe o des Gesetztes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

.....  
Ort/Datum

.....  
Unterschrift