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 qualtiy 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

Chapter 1

Implementation

In computer graphics, we generally synthesize 2d images from a given 3d scene description[1]. 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[2], having some material properties[3] 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[4], a cross-language, multiplattform 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 previouse 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 autum 2012[5].

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

[1]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.

[2]With the term world we are refering to a global coordinate system which is used in order to place all objects.

[3]Example material properties are: textures, surface colors, reflectance coefficients, refractive indices and so on.

[4]Official website:`http://www.opengl.org/`

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

[6]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`

lowing input data[7]:

- the structure of snake skin of different species[8] 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[9]. 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 Fransformations for a given input height field, representing a natural grating. For that purpose we have written a small Matlab [10] script conceptialized 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 $ifft2$[11]. 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.

[7]All data is provided by the Laboratory of Artificial and Natural Evolition in Geneva. See their website:`www.lanevol.org`

[8]We are using height field data for Elaphe and Xenopeltis snakes individuals like shown in figure **??**

[9]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

[10]Matlab is a interpreted scripting language which offers a huge collection of mathematical and numerically fast and stable algorithms.

[11]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[12] in the range $[0,1]$ and the range extrema are stored seperately 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 rescaled all frequency values in the DFT terms.

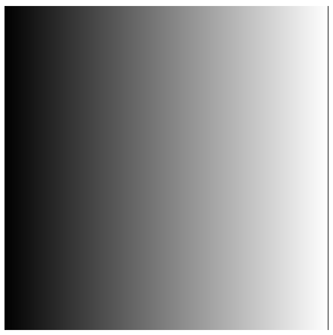
[12]We are talking about the i2dFFT of our height fields to the power of n. This is an N by N matrix (assuming the discrete height field was an N by N image), for which each component is a complex number. Hence, there is a a complex extrema as well as a imaginary extrema.

Algorithm 1 Precomputation: Pseudo code to generate Fourier terms

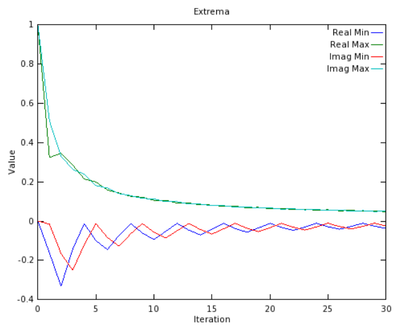
INPUT *heightfieldImg, maxH, dH, termCnt*
OUTPUT *DFT terms stored in Files*

```
% 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
```

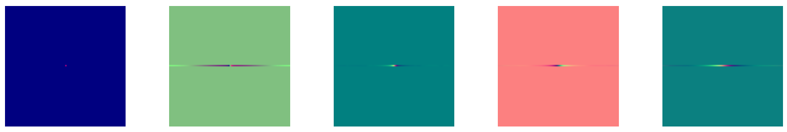
They key idea of algorithm 1 is to compute iteratively the Fourier Transformation for different powers of the provided height field. These DFT values are scaled by according to their extrema values. Another note about the command fftshift: It rearranges the output of the ifft2 by moving the zero frequency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



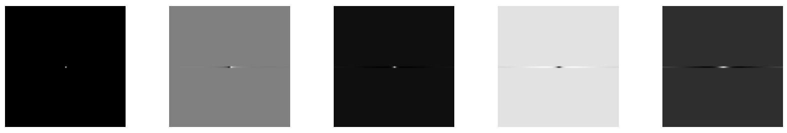
(a) Heightfield of a Blazed Grating



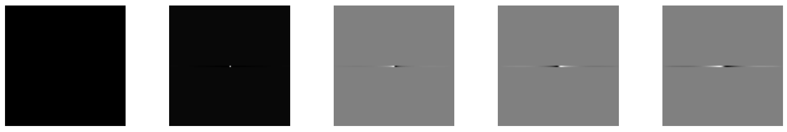
(b) Plot of extreme values for different powers of blazed grating



(c) ImRe0 (d) ImRe1 (e) ImRe4 (f) ImRe10 (g) ImRe20



(h) Re0 (i) Re1 (j) Re4 (k) Re10 (l) Re20



(m) Im0 (n) Im1 (o) Im4 (p) Im10 (q) Im20

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[13] 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 figur line from figure $1.1(m)$ until figure $1.1(q)$ showing the correspinding imaginary part DFT term images.

[13]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[14] 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.

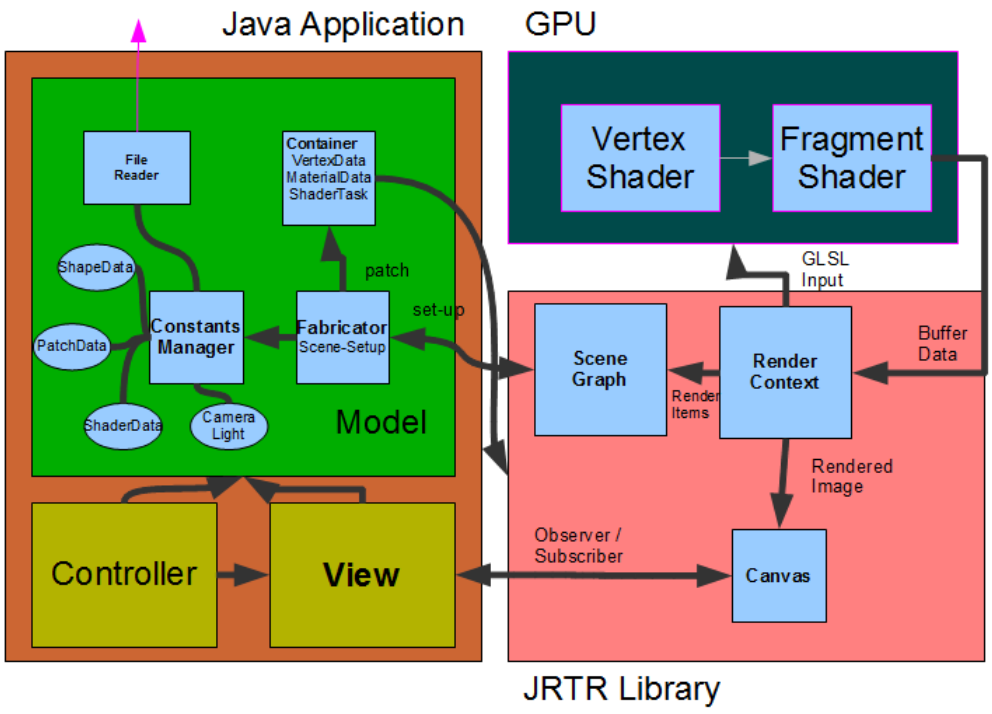
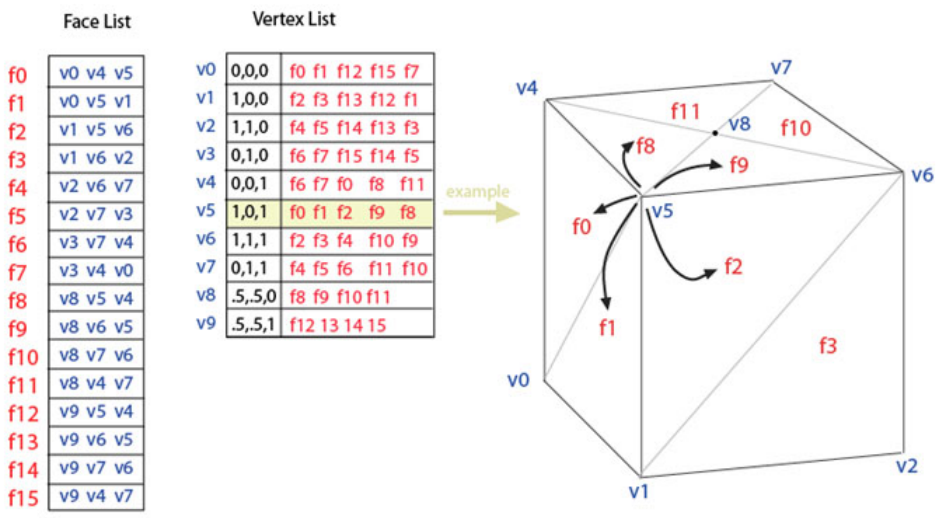


Figure 1.2: Schematical architecture of our Java renderer.

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[15] for a given height field[16] like visualized in figure 1.1. A shapes is a geometrical object defined by a triangular mesh as shown in figure 1.3.

[14]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.
[15]See section 1.1 for further information.
[16]and other height field constants such as the maximal height of its bumps or its pixel real-world width correspondance.



Face List

f0	v0 v4 v5
f1	v0 v5 v1
f2	v1 v5 v6
f3	v1 v6 v2
f4	v2 v6 v7
f5	v2 v7 v3
f6	v3 v7 v4
f7	v3 v4 v0
f8	v8 v5 v4
f9	v8 v6 v5
f10	v8 v7 v6
f11	v8 v4 v7
f12	v9 v5 v4
f13	v9 v6 v5
f14	v9 v7 v6
f15	v9 v4 v7

Vertex List

v0	0,0,0	f0 f1 f12 f15 f7
v1	1,0,0	f2 f3 f13 f12 f1
v2	1,1,0	f4 f5 f14 f13 f3
v3	0,1,0	f6 f7 f15 f14 f5
v4	0,0,1	f6 f7 f0 f8 f11
v5	1,0,1	f0 f1 f2 f9 f8
v6	1,1,1	f2 f3 f4 f10 f9
v7	0,1,1	f4 f5 f6 f11 f10
v8	.5,.5,0	f8 f9 f10 f11
v9	.5,.5,1	f12 13 14 15

Figure 1.3: Representation[17] 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 oder 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 rendern an image whereas the vertex shader is the first shading stage in the rendering pipeline.

[17]Modified image which originally has been taken from `http://en.wikipedia.org/wiki/Polygon_mesh`

As an input, a vertex shader 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[18] 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[19] of a vertex shader is to tranform the position of vertices. Notice that a vertex shader can manipulate the position of a vertices, but cannot generate additional mesh vertices. Therfore, 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.

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:

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

where P, C^{-1} and M are transformation matrices[20], defined the following way:

Model matrix M: Each vertex position of a shape is initially defined in a local coordinate system. To make is 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 1.4. 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}.

Frustum P: The Matrix P defines a perspective projection onto image plance, i.e. for any given position in camera space, P determines the corresponding 2d image coordinate. Persepctive 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 1.4, 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.

[18]Defining the positions of a shape in a local coordinate system simplifies its modeling process and allows us to apply transformations to a shape.

[19]Furthermore, texture coordinates used for texture-lookup within the fragment shader and per vertex lightning can be computed.

[20]These tranformation matrices are linear transformations expressed in homogenous coordinates.

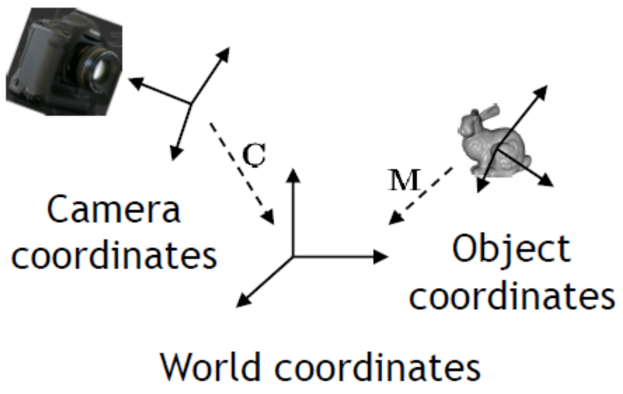


Figure 1.4: Illustration[21] of the Camera coordinate system where its origin defines the center of projection of camera.

The camera matrix C is constructed from its center of projection e, the position d where the cameras 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 1.5 illustrates geometrical setup required in order to construct C.

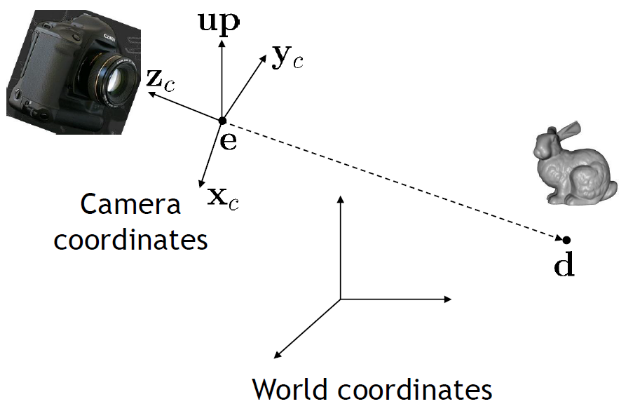


Figure 1.5: Illustration[22] of involved components in order to construct the camera matrix C. The eye-vector e denotes the positon of the camera in space, d is the position the camera looks at, and up denotes the cameras height. The camera space is spanned by the vectors helper vectors x_c, y_c and z_c. Notice that objects we look at are in front of us, and thus have negative z values

The mathematical representation of these vectors, x_c, y_c and z_c, spanning the camera space, introduced in figure 1.5, looks like the the following:

[21]This image has been taken from the lecture slides of computer graphics class 2012 which can be found on ilias.
[22]This image has been taken from the lecture slides of computer graphics class 2012 which can be found on ilias.

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

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:

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

In our vertex shader, besides transforming the vertex positions like described in equation 1.2, for every vertex, we also compute the direction vectors ω_i and ω_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 D.3. 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:     $p_{per} = P \cdot C^{-1} \cdot M \cdot p_{obj}$
12: end for
```

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 1.6.

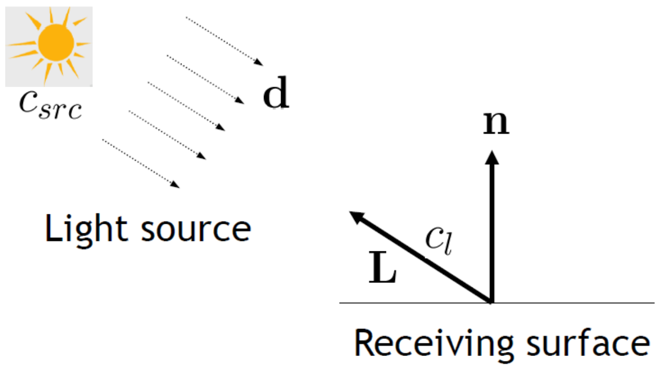


Figure 1.6: Illustration[23] of our light source setup. For a directional light source, all light rays are in parallel.

Last, in order to transform the positions of our mesh like described in equation 1.2, we also have to pass these transformation matrices[24]. 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 spanes 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 ω_i and viewing direction ω_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

The fragment shader is the OpenGL pipeline stage after a primitive is rasterized

vertices span a triangle each vertex position is transformed onto the image plane

The fragment shader on the other hand takes care of how the pixels between the vertices look. They are interpolated between the defined vertices following specific rules. where the image is calculated and the pixels between the vertices are filled in or coloured.

The term fragment is used because rasterization breaks up each geometric primitive, such as a triangle, into pixel-sized fragments for each pixel that the primitive covers. A fragment has an associated pixel location, a depth value, and a set of interpolated parameters such as a color, a secondary (specular) color, and one or more texture coordinate sets. These various interpolated parameters are derived from the transformed vertices that make up the particular geometric primitive used to generate the fragments

output a depth value, either written by the fragment shader or passed through from the screen-space fragment's Z value.

output produced by rasterizer

[23]This image has been taken from the lecture slides of computer graphics class 2012 which can be found on ilias.
[24]When speaking about transformation matrices, we are refering to the model, camera and frustum matrix.

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 spaning vertices, each previousely 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 computes position of a pixel is denoted as a fragment. Figure 1.7 conceptualizes the idea of processed set of fragment computed by a rasterizer.

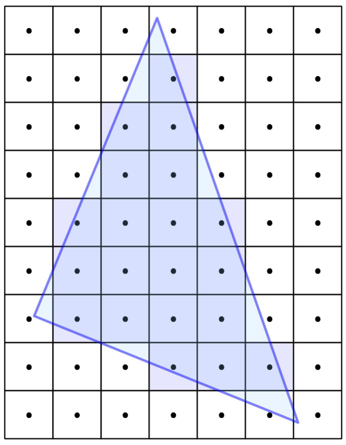


Figure 1.7: Illustration[25] of fragments covered by mesh triangle computed by the OpengGL rasterizer.

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 ω_i and ω_r defined like in figure **??**. In this section we explain how the color values of the structural colors resulting from our BRDF model are computed by a fragment shader.

Fur this purpose using the precompute DFT terms like explain in section 1.1 from section *d* and the vertex shader output from section.

Explain how we can compute

1.3.1

The purpose of a fragment shader is to render per fragment. A fragment is spanned by three vertices of a given mesh. For each pixel within a fragment in the fragment shader, the output of from its spanning vertices computed in the vertex shaders 2 is trilinearly interpolated depending on the pixel's position within the fragment. Furthermore, there can be additional input be assigned which is not directly interpolated from the output of vertex shader programs. In our fragment shader 3 this will be: all the references to the image buffers, containing the Fourier images computed in Matlab 1.1, the number steps for the taylor approximation (in our shader 30), the minimal and maximal wavelength, scaling factors, a reference to a lookup table containing the

[25]This image was taken from `http://en.wikibooks.org/wiki/GLSL_Programming/Rasterization`

CIE_{XYZ} color weights. Basically the whole computation within our fragment shader relies on the direction of light and the viewing direction. Our shader performs a numerical integration for our final derived expression in equation **??** using the trapezoidal-rule with uniform discretization of the wavelength spectrum at $5nm$ step sizes. This implies we are compressing sampled frequencies to the region near to the origin of their frequency domain due to the fact we are dividing the (u, v) by the wavelength and this implies that the (u, v) space is sampeled non-linearly.

The Gaussian window approach derived in section **??** is performed for each discrete λ value using a window large enough to span $4\sigma_f$ in both dimensions. For precomputing DFT tables we generally use nanostructure height fields that span at least $65\mu m^2$ and are sampled with resolution of at least 100nm. This ensures that the spectral response encompasses all the wavelengths in the visible spectrum, i.e. from 380nm to 780nm.

Algorithm 3 Fragment diffraction shader pseudo code

Input: Precomputed DFT Terms
Mesh Triangles
ω_i and ω_r
Output: Structural Color of a pixel

Procedures: the LIST

```
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:          $xyzWeights = ColorWeights(\lambda)$
6:          $lookupCoord = lookupCoord(u, v, \lambda)$
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:                     $pos = localLookUp(i, j, n)$
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
```

From line 4 to 26:
This loop performs uniform sampling along wavelength-space. $ColorWeights(\lambda)$ computes the color weight for the current wavelength λ by linear interpolation between the color weight for $\lceil \lambda \rceil$ and $\lfloor \lambda \rfloor$ which are stored in a external weights-table (assuming this table contains wavelengths in 1nm steps). At line 6: $lookupCoord(u, v, \lambda)$ the coordinates for the texture lookup are computed - See 1.7. Line 25 sums up the diffraction color contribution for the current wavelength in iteration λ.

From line 9 to 24:
This loop performs the Taylor series approximation using T terms. Basically, the spectral response is approximated for our current (u, v, λ). Furthermore, neighborhood boundaries for the gaussian-window sampling are computed, denoted as anchorX and anchorY.

From line 14 to 22:
In this inner most loop, the convolution of the gaussian window with the DFT of the patch is performed. $gaussWeightOf(dist)$ computes the weights in equation (**??**) from the distance between the current pixel's coordinates and the current neighbor's position in texture space. Local lookup coordinates for the current fourier coefficient $fftVal$ value are computed at line 17 and computed like described in 1.9. The actual texture lookup is performed at line 18 using those local coordinates. Inside $rescaledFourierValueAt$ the values $fftVal$ is rescaled by its extrema, i,e. $(fftVal * Max + Min)$ is computed, since $fftVal$ is normalized 1.1. The current $fftVal$ values in iteration is scaled by the current gaussian weight and then summed to the final neighborhood FFT contribution at line 20.

After line 26:
At line 27 the gain factor $C(\omega_i, \omega_r)$ **??** is multiplied by the current computed pixel color like formulated in **??**. The gain factor contains the geometric term $refeq : geometricterm$ and the Fresnel term F. W approximate F by the Schlick approximation $D.1$, using an reactive index at 1.5 since this is close to the measured value from snake sheds. Our BRDF values are scaled by s shadowing function as described in (SEE REFERENCES - PAPER), since most of the grooves in the snake skin nano-structures would from a V-cavity along the plane for a wave front with their top-edges at almost the same height.

Last, we transform our colors from the CIE_XYZ colorspace to the CIE_RGB space using the CIE Standard Illuminant D65, followed by a gamma correction. See 1.4.3 for further insight.

1.4 Technical details

1.4.1 Texture lookup

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 the bottom left corner of an image has the coordinates $(0, 0)$, whereas the top right corner has the value $(1, 1)$ assigned.

Given a nano-scaled surface patch P with a resolution A by A microns stored as an N by N pixel image I. Then one pixel in any direction corresponds to $dH = \frac{A}{N}\mu m$. In Matlab we compute a series of n output images $\{I_{out_1}, ..., I_{out_n}\}$ from I, which we will use for the lookup in our shader - See figure 1.8. For the lookup we use scaled and shifted (u, v) coordinates from **??**.

Since the zero frequency component of output images was shifted towards the centre of each image, we have to shift u, v to the center of the current N by N pixel image by a bias b. Mathematically, the bias is a constant value is computed the following:

$$b = (N\%2 == 0) \quad ? \quad \frac{N}{2} : \frac{N-1}{2} \tag{1.5}$$

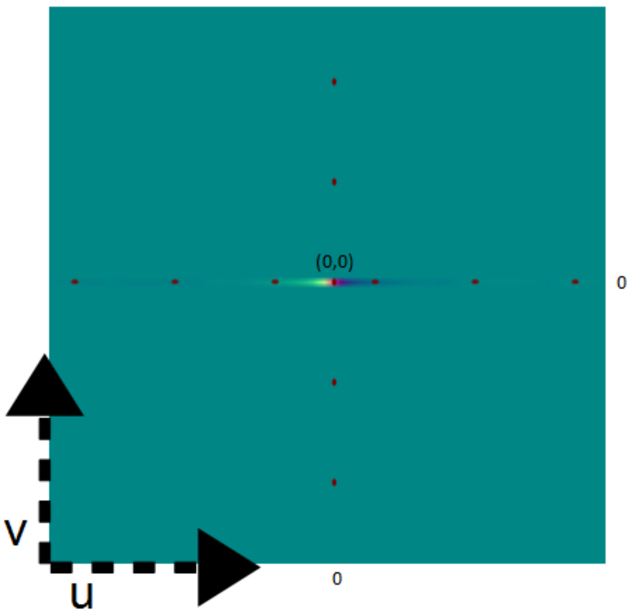


Figure 1.8: (u, v) lookup image

For the scaling we have to think a little further: lets 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 $w0 = 2pi/T$ and time interval $t0$. Let $k = \frac{2\pi}{\lambda}$ denote the wavenumber for the current wavelength λ. Then the signal is both periodic with time period T and discrete with time interval t_0 then its spectrum should be both discrete with frequency interval w_0 and periodic with frequency period $\Omega = \frac{2\pi}{t_0}$. This gives us the idea how to discretize the spectrum: Let us consider our Patch P assuming it is distributed as a periodic function on our surface. Then, its frequency interval along the x direction is $w_0 = \frac{2\pi}{T} = \frac{2\pi}{N*dH}$. Thus only wave numbers 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:

$$\frac{ku}{w_0} = \frac{kuNdH}{2\pi} \tag{1.6}$$

$$= \frac{uNdH}{\lambda} \tag{1.7}$$

Using those findings 1.5, 1.7, the final (u, v) texture lookup-coorinates for the current wavelength λ in iteration, will then look like:

$$(u_{lookup}, v_{lookup}) = \left(\frac{uNdH}{\lambda} + b, \frac{vNdH}{\lambda} + b\right) \tag{1.8}$$

Note for the windowing approach we are visiting a one pixel neighborhood for each pixel p. This is like a base change with (u_{lookup}, v_{lookup}) as new coordinate system origin. The lookup coordinates for the neighbor-pixel (i, j) are:

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

1.4.2 Texture Blending

The final rendered color for each pixel 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 $w_{diffuse}$. the texture color is once scales by a binary weight determined by the absolute value of the Fresnel Term F and once by $1 - w_{diffuse}$.

Algorithm 4 Texture Blending

```
$\alpha = (abs(F) > 1)?1 : 0$
$c_{out} = (1 - w_{diffuse}) * c_{diffraction} + (1 - \alpha) * c_{texture} + w_{diffuse} * c_{texture})$
```

1.4.3 Color Transformation

In our shader we access a table which contains precomputed CIE's color matching functions values from $\lambda_{min} = 380nm$ to $\lambda_{max} = 780nm$ in $5nm$ steps. Such a function value table can be found at downloaded at cvrl.ioo.ucl.ac.uk for example. We compute the (X, Y, Z) CIE_{XYZ} color values as described in section ??.

We can transform the color values into CIE_{RGB} by performing the following linear transformation:

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

where one possible transformation is:

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

There are some other color space transformation. The shader uses the CIE Standard Illuminant D65 which is intended to represent average daylight. Using D65 the whole colorspace transformation will look like:

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

Last we perfrom gamma correction on each pixel's (R, G, B) value. Gamma correction is a non linear transformation which controls the overall brightness of an image.

1.5 Discussion

The fragment shader algorithm described in 3 performs the gaussian window approach by sampling over the whole wavelength spectrum in uniform step sizes. This algorithm is valid but also slow since we iterate for each pixel over the whole lambda spectrum. Furthermore, for any pixel, we iterate over its 1 neighborhood. Considering the loop for the taylor approximation as well, we will have a run-time complexity of $O(\#spectrtumIter \cdot \#taylorIter \cdot neighborhoodRadius^2)$. Hence, Instead sampling over the whole wavelength spectrum, we could instead integrate over just a few required lambdas which are elicited like the following: Lets consider (u, v, w) defined as ??. 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 $\lambda_n^v = \frac{du}{n}$. For $n = 0$ there it follows $(u, v) = (0, 0)$. If $u, v > 0$

$$N_{min}^u = \frac{du}{\lambda_{max}} \leq n_u \leq \frac{du}{\lambda_{min}} = N_{min}^u$$
$$N_{min}^v = \frac{dv}{\lambda_{max}} \leq n_v \leq \frac{dv}{\lambda_{min}} = N_{min}^v$$

If $u, v < 0$

$$N_{min}^u = \frac{du}{\lambda_{min}} \leq n_u \leq \frac{du}{\lambda_{min}} = N_{max}^u$$
$$N_{min}^v = \frac{dv}{\lambda_{min}} \leq n_v \leq \frac{dv}{\lambda_{min}} = N_{max}^v$$

By transforming those equation to $(\lambda_{min}^u, \lambda_{min}^u)$, $(\lambda_{min}^v, \lambda_{min}^v)$ respectively for any (u, v, w) for each pixel we can reduce the total number of required iterations in our shader.

Another variant is the PQ approach described in chapter 2 **??**. Depending on the interpolation method, there are two possible variants we can think of as described in **??**. Either we try to interpolate linearly or use sinc interpolation. The first variant does not require to iterate over a pixel's neighborhood, it is also faster than the gaussian window approach. One could think of a combination of those tho optimization approaches. Keep in mind, both of these approaches are further approximation. The quality of the rendered images will suffer using those two approaches. The second variant, 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 neighborhood within the fragment shader which will slow down the whole shading. The following algorithm describes the modification of the fragment shader 3 in oder to use sinc interpolation for the PQ approach **??**.

Algorithm 5 Sinc interpolation for PQ approach

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

In a fragment shader we compute for each pixel p in the current fragment its reconstructed function value $f(p)$ stores in w_p. w_p is the reconstructed signal value at $f(p)$ by the sinc function as described in **??**. We calculate the distance $\Delta_{p,(i,j)}$ between the current pixel p and each of its neighbor pixels $(i, j) \in \mathcal{N}_1(p)$ in its one-neighborhood. Multiplying this distance by π gives us the an angle used for the sinc function interpilation. We add a small integer ϵ in order to avoid division by zeros side-effects.

Appendix A

Signal Processing Basics

A signal is a function that conveys information about the behavior or attributes of some phenomenon. In the physical world, any quantity exhibiting variation in time or variation in space (such as an image) is potentially a signal that might provide information on the status of a physical system, or convey a message between observers.

The Fourier Transform is an important image processing tool which is used to decompose an image into its sine and cosine components. The output of the transformation represents the image in the Fourier or frequency domain, while the input image is the spatial domain equivalent. In the Fourier domain image, each point represents a particular frequency contained in the spatial domain image.

A.1 Fourier Transformation

The Fourier-Transform is a mathematical tool which allows to transform a given function or rather a given signal from defined over a time- (or spatial-) domain into its corresponding frequency-domain.

Let f an measurable function over $\mathbb{R}^n$. Then, the continuous Fourier Transformation(**FT**), denoted as $\mathcal{F}\{f\}$ of f, ignoring all constant factors in the formula, is defined as:

$$\mathcal{F}_{FT}\{f\}(w) = \int_{\mathbb{R}^n} f(x) e^{-iwt} dt \tag{A.1}$$

whereas its inverse transform is defined like the following which allows us to obtain back the original signal:

$$\mathcal{F}_{FT}^{-1}\{f\}(w) = \int_{\mathbb{R}} \mathcal{F}\{w\} e^{iwt} dt \tag{A.2}$$

Usual w is identified by the angular frequency which is equal $w = \frac{2\pi}{T} = 2\pi v_f$. In this connection, T is the period of the resulting spectrum and v_f is its corresponding frequency.

By using Fourier Analysis, which is the approach to approximate any function by sums of simpler trigonometric functions, we gain the so called Discrete Time Fourier Transform (in short **DTFT**). The DTFT operates on a discrete function. Usually, such an input function is often created by digitally sampling a continuous function. The DTFT itself is operation on a discretized signal on a continuous, periodic frequency domain and looks like the following:

$$\mathcal{F}_{DTFT}\{f\}(w) = \sum_{-\infty}^{\infty} f(x)e^{-iwk} \tag{A.3}$$

Note that the DTFT is not practically suitable for digital signal processing since there a signal can be measured only in a finite number of points. Thus, we can further discretize the frequency domain and will get then the Discrete Fourier Transformation (in short **DFT**) of the input signal:

$$\mathcal{F}_{DFT}\{f\}(w) = \sum_{n=0}^{N-1} f(x)e^{-iw_nk} \tag{A.4}$$

Where the angular frequency w_n is defined like the following $w_n = \frac{2\pi n}{N}$ and N is the number of samples within an equidistant period sampling.

Any continuous function $f(t)$ can be expressed as a series of sines and cosines. This representation is called the Fourier Series (denoted by FS) of $f(t)$.

$$f(t) = \frac{1}{2}a_0 + \sum_{n=1}^{\infty} a_n cos(nt) + \sum_{n=1}^{\infty} b_n cos(nt) \tag{A.5}$$

where

$$\begin{aligned} a_0 &= \int_{-\pi}^{\pi} f(t)dt \\ a_n &= \frac{1}{\pi}\int_{-\pi}^{\pi} f(t)cos(nt)dt \\ b_n &= \frac{1}{\pi}\int_{-\pi}^{\pi} f(t)sin(nt)dt \end{aligned} \tag{A.6}$$

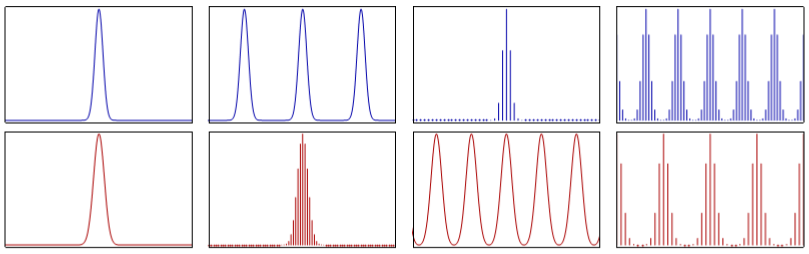


Figure A.1: Relationship[1] between the continuous Fourier transform and the discrete Fourier transform: Left column: A continuous function (top) and its Fourier transform $A.1$ (bottom). Center-left column: Periodic summation of the original function (top). Fourier transform (bottom) is zero except at discrete points. The inverse transform is a sum of sinusoids called Fourier series $A.5$. Center-right column: Original function is discretized (multiplied by a Dirac comb) (top). Its Fourier transform (bottom) is a periodic summation (DTFT) of the original transform. Right column: The DFT $A.4$ (bottom) computes discrete samples of the continuous DTFT $A.3$. The inverse DFT (top) is a periodic summation of the original samples.

[1]image of illustration has been taken from wikipedia

Spetail signal $f(t)$ is	Operator	Transformed frequency signal $\hat{f}(\omega)$ is
continuous and periodic in t	FS $A.5$	only discrete in ω
only continuous in t	FT $A.1$	only continuous in ω
only discrete in t	DTFT $A.3$	continuous and periodic in ω
discrete and periodic in t	DFT $A.4$	discrete and periodic in ω

Table A.1: Fourier operator to apply for a given spatial input signal and the properties of its resulting output signal in frequency space

A.2 Convolution

The convolution $f * g$ of two functions f, $g\colon \mathbb{R}^n \to \mathbb{C}$ is defined as:

$$(f * g)(t) = \int_{\mathbb{R}^n} f(t)g(t - x)dx \tag{A.7}$$

Note that the Fourier transform of the convolution of two functions is the product of their Fourier transforms. This is equivalent to the fact that Convolution in spatial domain is equivalent to multiplication in frequency domain. Therefore, the inverse Fourier transform of the product of two Fourier transforms is the convolution of the two inverse Fourier transforms. Last an illustration of the relationships between the previous presented Fourier transformations and different given input signals. First an concrete example shown in Figure $A.1$. Table $A.1$ tells what Fourier transformation operator has to be applied to which kind of input signal and what properties its resulting Fourier transform will have.

A.3 Taylor Series

Taylor series is a representation of a function as an infinite sum of terms that are calculated from the values of the function's derivatives at a single point.

The Taylor series $\mathcal{T}$ of a real or complex-valued function $f(x)$ that is infinitely differentiable at a real or complex number a is the power series:

$$\mathcal{T}(f;a)(x) = \sum_{n=0}^{\infty} \frac{f^n(a)}{n!}(x - a)^n \tag{A.8}$$

Appendix B

Summary of Stam's Derivations

In his paper about Diffraction Shader, J. Stam derives a BRDF which is modeling the effect of diffraction for various analytical anisotropic reflexion models relying on the so called scalar wave theory of diffraction for which a wave is assumed to be a complex valued scalar. It's noteworthy, that Stam's BRDF formulation does not take into account the polarization of the light. Fortunately, light sources like sunlight and light bulbs are unpolarized.

A further assumption in Stam's Paper is, the emanated waves from the source are stationary, which implies the wave is a superposition of independent monochromatic waves. This implies that each wave is associated to a definite wavelength lambda. However, sunlight once again fulfills this fact.

In our simulations we will always assume we have given a directional light source, i.e. sunlight. Hence, Stam's model can be used for our derivations.

For his derivations Stam uses the Kirchhoff integral[1], which is relating the reflected field to the incoming field. This equation is a formalization of Huygen's well-known principle that states that if one knows the wavefront at a given moment, the wave at a later time can be deduced by considering each point on the first wave as the source of a new disturbance. Mathematically speaking, once the field $\psi_1 = e^{ik\mathbf{x}\cdot\mathbf{ss}}$ on the surface is known, the field ψ_2 everywhere else away from the surface can be computed. More precisely, we want to compute the wave ψ_2 equal to the reflection of an incoming planar monochromatic wave $\psi_1 = e^{ik\omega_i * x}$ traveling in the direction ω_i from a surface S to the light source. Formally, this can be written as:

$$\psi_2(\omega_i, \omega_r) = \frac{ike^{iKR}}{4\pi R}(F(-\omega_i - \omega_r) - (-\omega_i + \omega_r)) \cdot I_1(\omega_i, \omega_r) \tag{B.1}$$

with

$$I_1(\omega_i, \omega_r) = \int_S \hat{\mathbf{n}} e^{ik(-\omega_i - \omega_r)\cdot \mathbf{s} d\mathbf{s}} \tag{B.2}$$

In applied optics, when dealing with scattered waves, one does use differential scattering cross-section rather than defining a BRDF which has the following identity:

$$\sigma^0 = 4\pi \lim_{R\to\infty} R^2 \frac{\langle |\psi_2|^2 \rangle}{\langle |\psi_1|^2 \rangle} \tag{B.3}$$

where R is the distance from the center of the patch to the receiving point x_p, $\hat{\mathbf{n}}$ is the normal of the surface at s and the vectors:

[1]See `http://en.wikipedia.org/wiki/Kirchhoff_integral_theorem` for further information.

The relationship between the BRDF and the scattering cross section can be shown to be equal to

$$BRDF = \frac{1}{4\pi}\frac{1}{A}\frac{\sigma^0}{cos(\theta_i)cos(\theta_r)} \tag{B.4}$$

where θ_i and θ_r are the angles of incident and reflected directions on the surface with the surface normal n. See **??**.

The components of vector resulting by the difference between these direction vectors: In order to simplify the calculations involved in his vectorized integral equations, Stam considers the components of vector

$$(u, v, w) = -\omega_i - \omega_r \tag{B.5}$$

explicitly and introduces the equation:

$$I(ku, kv) = \int_S \hat{\mathbf{n}} e^{ik(u,v,w)\cdot \mathbf{s} d\mathbf{s}} \tag{B.6}$$

which is a first simplification of $B.2$. Note that the scalar w is the third component of **??** and can be written as $w = -(cos(\theta_i) + cos(\theta_r))$ using spherical coordinates. The scalar $k = \frac{2\pi}{\lambda}$ represent the wavenumber.

During his derivations, Stam provides a analytical representation for the Kirchhoff integral assuming that each surface point $s(x, y)$ can be parameterized by $(x, y, h(x, y))$ where h is the height at the position (x, y) on the given (x, y) surface plane. Using the tangent plane approximation for the parameterized surface and plugging it into $B.6$ he will end up with:

$$\mathbf{I}(ku, kv) = \int\int (-h_x(x, y), -h_y(x, y), 1) e^{ikwh(x,y)} e^{ik(ux+vy)} dxdy \tag{B.7}$$

For further simplification Stam formulates auxillary function which depends on the provided height field:

$$p(x, y) = e^{iwkh(x,y)} \tag{B.8}$$

which will allow him to further simplify his equation $B.7$ to:

$$\mathbf{I}(ku, kv) = \int\int \frac{1}{ikw}(-p_x, -p_y, ikwp) dxdy \tag{B.9}$$

where he used that $(-h_x(x, y), -h_y(x, y), 1)e^{kwh(x,y)}$ is equal to $\frac{(-p_x, -p_y, ikwp)}{ikw}$ using the definition of the partial derivatives applied to the function **??**.

Let $P(x, y)$ denote the Fourier Transform (FT) of $p(x, y)$. Then, the differentiation with respect to x respectively to y in the Fourier domain is equivalent to a multiplication of the Fourier transform by $-iku$ or $-ikv$ respectively. This leads him to the following simplification for $B.7$:

$$\mathbf{I}(ku, kv) = \frac{1}{w} P(ku, kv) \cdot (u, v, w) \tag{B.10}$$

Let us consider the term $g = (F(-\omega_i - \omega_r) - (-\omega_i + \omega_r))$, which is a scalar factor of $B.1$. The dot product with g and $(-\omega_i - \omega_r)$ is equal $2F(1 + \omega_i \cdot \omega_r)$. Putting this finding and the identity $B.10$ into $B.1$ he will end up with:

$$\psi_2(\omega_i, \omega_r) = \frac{ike^{iKR}}{4\pi R}\frac{2F(1 + \omega_i \cdot \omega_r)}{w} P(ku, kv) \tag{B.11}$$

By using the identity $B.4$, this will lead us to his main finding:

$$BRDF_{\lambda}(\omega_i, \omega_r) = \frac{k^2 F^2 G}{4\pi^2 A w^2} \langle |P(ku, kv)|^2 \rangle \tag{B.12}$$

where G is the so called geometry term which is equal:

$$G = \frac{(1 + \omega_i \cdot \omega_r)^2}{cos(\theta_i) cos(\theta_r)} \tag{B.13}$$

Appendix C

Derivation Steps in Detail

C.1 Taylor Series Approximation

For an $N \in \mathbb{N}$ such that

$$\sum_{n=0}^{N} \frac{(ikwh)^n}{n!} \mathcal{F}\left\{h^n\right\}(\alpha, \beta) \approx P(\alpha, \beta) \tag{C.1}$$

we have to prove:

1. Show that there exist such an $N \in \mathbb{N}$s.t the approximation holds true.
2. Find a value for B s.t. this approximation is below a certain error bound, for example machine precision ϵ.

C.1.1 Proof Sketch of 1.

By the **ratio test** (see [1]) It is possible to show that the series $\sum_{n=0}^{N} \frac{(ikwh)^n}{n!} \mathcal{F}\left\{h^n\right\}(\alpha, \beta)$ converges absolutely:

Proof: Consider $\sum_{k=0}^{\infty} \frac{y^n}{n!}$ where $a_k = \frac{y^k}{k!}$. By applying the definition of the ratio test for this series it follows:

$$\forall y : limsup_{k \to \infty} | \frac{a_{k+1}}{a_k} | = limsup_{k \to \infty} \frac{y}{k+1} = 0 \tag{C.2}$$

Thus this series converges absolutely, no matter what value we will pick for y.

C.1.2 Part 2: Find such an N

Let $f(x) = e^x$. We can formulate its Taylor-Series, stated above. Let $P_n(x)$denote the n-th Taylor polynom,

$$P_n(x) = \sum_{k=0}^{n} \frac{f^{(k)}(a)}{k!} (x-a)^k \tag{C.3}$$

where a is our developing point (here a is equal zero).

We can define the error of the n-th Taylor polynom to be $E_n(x) = f(x) - P_n(x)$. the error of the n-th Taylor polynom is difference between the value of the function and the Taylor polynomial This directly implies $|E_n(x)| = |f(x) - P_n(x)|$. By using the Lagrangian Error Bound it follows:

$$|E_n(x)| \leq \frac{M}{(n+1)!}|x-a|^{n+1} \tag{C.4}$$

with $a = 0$, where **M** is some value satisfying $|f^{(n+1)}(x)| \leq M$ on the interval $I = [a, x]$. Since we are interested in an upper bound of the error and since **a** is known, we can reformulate the interval as $I = [0, x_{max}]$, where

$$x_{max} = \|i\| k_{max} w_{max} h_{max} \tag{C.5}$$

We are interested in computing an error bound for $e^{ikwh(x,y)}$. Assuming the following parameters and facts used within Stam's Paper:

- Height of bump: 0.15micro meters
- Width of a bump: 0.5micro meters
- Length of a bump: 1micro meters
- $k = \frac{2\pi}{\lambda}$ is the wavenumber, $\lambda \in [\lambda_{min,} \lambda_{max}]$ and thus $k_{max} = \frac{2\pi}{\lambda_{min}}$. Since $(u, v, w) = -\omega_i - \omega_r$ and both are unit direction vectors, each component can have a value in range [-2, 2].
- for simplification, assume$[\lambda_{min,} \lambda_{max}] = [400nm, 700nm]$.

We get:

$$\begin{aligned} x_{max} &= \|i\| * k_{max} * w_{max} * h_{max} \\ &= k_{max} * w_{max} * h_{max} \\ &= 2 * (\frac{2\pi}{4 * 10^{-7} m}) * 1.5 * 10^{-7} \\ &= 1.5\pi \end{aligned} \tag{C.6}$$

and it follows for our interval $I = [0, 1.5\pi]$.

Next we are going to find the value for M. Since the exponential function is monotonically growing (on the interval I) and the derivative of the **exp** function is the exponential function itself, we can find such an M:

$$\begin{aligned} M &= e^{x_{max}} \\ &= exp(1.5\pi) \end{aligned}$$

and $|f^{(n+1)}(x)| \leq M$ holds. With

$$\begin{aligned} |E_n(x_{max})| &\leq \frac{M}{(n+1)!}|x_{max} - a|^{n+1} \\ &= \frac{exp(1.5\pi) * (1.5\pi)^{n+1}}{(n+1)!} \end{aligned} \tag{C.7}$$

we now can find a value of n for a given bound, i.e. we can find an value of $N \in \mathbb{N}$ s.t. $\frac{exp(1.5\pi)*(1.5\pi)^{N+1}}{(N+1)!} \leq \epsilon$. With Octave/Matlab we can see:

- if N=20 then $\epsilon \approx 2.9950 * 10^{-4}$
- if N=25 then $\epsilon \approx 8.8150 * 10^{-8}$
- if N=30 then $\epsilon \approx 1.0050 * 10^{-11}$

With this approach we have that $\sum_{n=0}^{25} \frac{(ikwh)^n}{n!} \mathcal{F}\left\{h^n\right\}(\alpha, \beta)$ is an approximation of $P(u,v)$ with error $\epsilon \approx 8.8150 * 10^{-8}$. This means we can precompute 25 Fourier Transformations in order to approximate P(u,v) having an error $\epsilon \approx 8.8150 * 10^{-8}$.

C.2 PQ approach

C.2.1 One dimensional case

Since our series is bounded, we can simplify the right-hand-side of equation **??**.

Note that e^{-ix} is a complex number. Every complex number can be written in its polar form, i.e.

$$e^{-ix} = cos(x) + isin(x) \tag{C.8}$$

Using the following trigonometric identities

$$\begin{aligned} cos(-x) &= cos(x) \\ sin(-x) &= -sin(x) \end{aligned} \tag{C.9}$$

combined with C.8 we can simplify the series **??** even further to:

$$\frac{1 - e^{iwT(N+1)}}{1 - e^{-iwT}} = \frac{1 - cos(wT(N+1)) + isin(wT(N+1))}{1 - cos(wT) + isin(wT)} \tag{C.10}$$

Equation C.10 is still a complex number, denoted as $(p + iq)$. Generally, every complex number can be written as a fraction of two complex numbers. This implies that the complex number $(p + iq)$ can be written as $(p + iq) = \frac{(a+ib)}{(c+id)}$ for any $(a + ib), (c + id) \neq 0$. Let us use the following substitutions:

$$\begin{aligned} a &:= 1 - cos(wT(N+1)) & b &= sin(wT(N+1)) \\ c &= 1 - cos(wT) & d &= sin(wT) \end{aligned} \tag{C.11}$$

Hence, using C.11, it follows

$$\frac{1 - e^{iwT(N+1)}}{1 - e^{-iwT}} = \frac{(a + ib)}{(c + id)} \tag{C.12}$$

By rearranging the terms, it follows $(a + ib) = (c + id)(p + iq)$ and by multiplying its right hand-side out we get the following system of equations:

$$\begin{aligned} (cp - dq) &= a \\ (dp + cq) &= b \end{aligned} \tag{C.13}$$

After multiplying the first equation of C.13 by c and the second by d and then adding them together, we get using the law of distributivity new identities for p and q:

$$\begin{aligned} p &= \frac{(ac+bd)}{c^2+d^2} \\ q &= \frac{(bc+ad)}{c^2+d^2} \end{aligned} \tag{C.14}$$

Using some trigonometric identities and putting our substitution from C.11 for a, b, c, d back into the current representation C.14 of p and q we will get:

$$\begin{aligned} p &= \frac{1}{2}+\frac{1}{2}\left(\frac{cos(wTN)-cos(wT(N+1))}{1-cos(wT)}\right) \\ q &= \frac{sin(wT(N+1))-sin(wTN)-sin(wT)}{2(1-cos(wT))} \end{aligned} \tag{C.15}$$

Since we have seen, that $\sum_{n=0}^{N} e^{-uwnT}$ is a complex number and can be written as $(p+iq)$, we now know an explicit expression for p and q. Therefore, the one dimensional inverse Fourier transform of S is equal:

$$\begin{aligned} \mathcal{F}^{-1}\{S\}(w) &= \mathcal{F}^{-1}\{f\}(w)\sum_{n=0}^{N} e^{-iwnT} \\ &= (p+iq)\mathcal{F}^{-1}\{f\}(w) \end{aligned} \tag{C.16}$$

C.2.2 Two dimensional case

$$\begin{aligned} \mathcal{F}^{-1}\{S\}(w_1,w_2) &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\sum_{n_2=0}^{N_1}\sum_{n_2=0}^{N_2} h(x_1+n_1T_1,x_2+n_2T_2)e^{iw(x_1+x_2)}dx_1dx_2 \\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\sum_{n_2=0}^{N_1}\sum_{n_2=0}^{N_2} h(y_1,y_2)e^{iw((y_1-n_1T_1)+(y_2+n_2T_2))}dx_1dx_2 \\ &= \sum_{n_2=0}^{N_1}\sum_{n_2=0}^{N_2}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} h(y_1,y_2)e^{iw(y_1+y_2)}e^{-iw(n_1T_1+n_2T_2)}dy_1dy_2 \\ &= \sum_{n_2=0}^{N_1}\sum_{n_2=0}^{N_2} e^{-iw(n_1T_1+n_2T_2)}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} Box(y_1,y_2)e^{iw(y_1+y_2)}dy_1dy_2 \\ &= \left(\sum_{n_2=0}^{N_1}\sum_{n_2=0}^{N_2} e^{-iw(n_1T_1+n_2T_2)}\right)\mathcal{F}^{-1}\{h\}(w_1,w_2) \\ &= \left(\sum_{n_2=0}^{N_1} e^{-iwn_1T_1}\right)\left(\sum_{n_2=0}^{N_2} e^{-iwn_2T_2}\right)\mathcal{F}^{-1}\{h\}(w_1,w_2) \\ &= (p_1+iq_1)(p_2+iq_2)\mathcal{F}^{-1}\{h\}(w_1,w_2) \\ &= ((p_1p_2-q_1q_2)+i(p_1p_2+q_1q_2))\mathcal{F}^{-1}\{h\}(w_1,w_2) \\ &= (p+iq)\mathcal{F}_{DTFT}\{h\}(w_1,w_2) \end{aligned} \tag{C.17}$$

Where we have defined

$$\begin{aligned} p &:= (p_1 p_2 - q_1 q_2) \\ q &:= (p_1 p_2 + q_1 q_2) \end{aligned} \tag{C.18}$$

Appendix D

Miscellaneous Transformations

D.1 Fresnel Term - Schlick's approximation

The Fresnel's equations describe the reflection and transmission of electromagnetic waves at an interface. That is, they give the reflection and transmission coefficients for waves parallel and perpendicular to the plane of incidence. Schlick's approximation is a formula for approximating the contribution of the Fresnel term where the specular reflection coefficient R can be approximated by:

$$R(\theta) = R_0 + (1 - R_0)(1 - \cos\theta)^5 \tag{D.1}$$

and

$$R_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2$$

where θ is the angle between the viewing direction and the half-angle direction, which is halfway between the incident light direction and the viewing direction, hence $\cos\theta = (H \cdot V)$. And n_1, n_2 are the indices of refraction of the two medias at the interface and R_0 is the reflection coefficient for light incoming parallel to the normal (i.e., the value of the Fresnel term when $\theta = 0$ or minimal reflection). In computer graphics, one of the interfaces is usually air, meaning that n_1 very well can be approximated as 1.

D.2 Spherical Coordinates and Space Transformation

$\forall \begin{pmatrix} x \\ y \\ z \end{pmatrix} \in \mathbb{R}^3 : \exists r \in [0, \infty) \exists \phi \in [0, 2\pi] \exists \theta \in [0, \pi]$ s.t.

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} rsin(\theta)cos(\phi) \\ rsin(\theta)sin(\phi) \\ rcos(\theta) \end{pmatrix}$$

From the definition **??** of $(u, v, w) = -\omega_i - \omega_r$ and using spherical coordinates $D.2$, we get for w the following identity:

$$\begin{aligned} w &= -\omega_i - \omega_r \\ &= -(\omega_i + \omega_r) \\ &= -\left(cos(\theta_i) + cos(\theta_r)\right) \end{aligned} \tag{D.2}$$

and therefore w^2 is equal $(cos(\theta_i) + cos(\theta_r))^2$.

D.3 Tangent Space

The concept of tangentspace-transformation of tangent space is used in order to convert a point between world and tangent space. GLSL fragment shaders require normals and other vertex primitives declared at each pixel point, which mean that we have one normal vector at each texel and the normal vector axis will vary for every texel.

Think of it as a bumpy suface defined on a flat plane. If those normals were declared in the world space coordinate system, we would have to rotate these normals every time the model is rotated, even when just for a small amount. Since the lights, cameras and other objects are usually defined in world space coordinate system, and therefore, when they are involved in an calculation within the fragment shader, we would to have to rotate them as well for every pixel. This would involve almost countless many object to world matrix transformations meed to take place at the pixel level. Therefore, instead doing so, we transform all vertex primitives into tangent space within the vertex shader.

To make this point clear an example: Even we would rotate the cube in figure $D.1$, the tangent space axis will remain aligned with respect to the face. Which practically speaking, will save us from performing many space transformations applied pixel-wise within the fragment shader and instead allows us to perform us the tangenspace transformation of every involved vertex primitive in the vertex-shader.

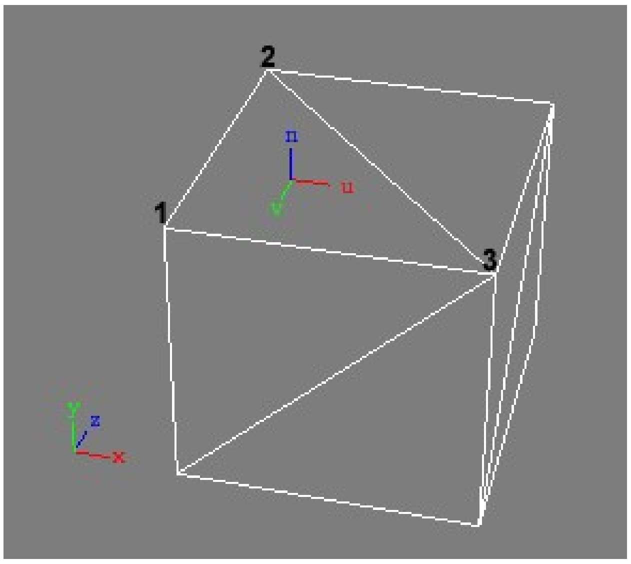


Figure D.1: Cube in world space (x, y, z) showing the tangen space (u, v, n) of its face $(2, 1, 3)$

List of Tables

List of Figures

List of Algorithms

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

[DSD14] D. S. Dhillon, et a.: Interactive Diffraction from Biological Nanostructures. In: *EUROGRAPHICS 2014/ M. Paulin and C. Dachsbacher* (2014), January

[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: Structurral 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