CHALMERS
UNIVERSITY OF TECHNOLOGY

Photorealistic 3D Rendering of Furniture

Implementation of Tools for Modeling, Shading and Rendering

Bachelor of Science Thesis in Computer Science and Engineering

SVEN HELLSTEN
JAKOB LARSSON
PETER MCEVOY
VIDAR NELSON
VICTOR OLAUSSON
JONATAN SOLIN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden, June 2016

Bachelor of Science Thesis in Computer Science and Engineering
2016:25

Photorealistic 3D Rendering of Furniture

Implementation of Tools for Modeling, Shading and Rendering

SVEN HELLSTEN
JAKOB LARSSON
PETER MCEVOY
VIDAR NELSON
VICTOR OLAUSSON
JONATAN SOLIN

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2016

Photorealistic 3D Rendering of Furniture
Implementation of Tools for Modeling, Shading and Rendering

Sven Hellsten
Jakob Larsson
Peter McEvoy
Vidar Nelson
Victor Olausson
Jonatan Solin

Supervisor: MARCO FRATARCANGELI
Examiner: OLOF TORGERSSON

Bachelor of Science Thesis in Computer Science and Engineering 2016:25
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A rattan chair and a pillow made of woven cloth rendered in OptiX, showcasing the three tools developed in this report.

Gothenburg, Sweden 2016

Abstract

Photorealistic 3D rendering of product images is successfully used by companies such as IKEA Communications AB. However, there are still problems where the current methods fall short. We investigate three such problems. First, we develop a tool for the 3D-modelling of woven rattan furniture. The resulting tool alleviates some of the work involved in creating detailed 3D models of rattan furniture by automatically generating an interwoven structure of strands. The tool falls short when attempting to create a heavily curved rattan mesh. Second, we implement a physically based shading model for woven cloth. The implemented shader can accurately simulate the appearance of certain types of woven cloth. Fabrics featuring more variation in structure prove difficult to recreate using this model. Finally, we develop a ray tracing renderer which uses hardware acceleration for the creation of images at interactive rates. This results in a tool which can preview scenes featuring complex objects and different materials, and serves as a proof-of-concept for utilizing hardware acceleration for ray tracing.

Sammanfattning

3D-rendering kan användas för att skapa realistiska produktbilder med hjälp av datorberäkningar och är en metod som bland annat tillämpas av IKEA Communications AB, som ett komplement till fotografering. Det finns fortfarande svårigheter med att skapa övertygande bilder helt digitalt och vi undersöker tre problem som är speciellt relevanta för möbelindustrin. Det första problemet är att skapa 3D-modeller av rottingmöbler, eftersom de har en väldigt komplex struktur. För att underlätta utformandet av denna typ av 3D-modeller utvecklar vi ett verktyg som kan skapa den sammanvävda geometrin automatiskt. Verktyget fungerar bra för enklare former men brister vid alltför kurvade rottingstrukturer. Det andra problemet är att simulera utseendet av vävt tyg på ett detaljerat vis. Vi implementerar en model som grundar sig på fysikaliska principer av hur ljuset faller på de individuella trådarna i tyget och vår implementation utvärderas genom att jämföra resultatet med fotografier av verkligt tyg. För textilier med regelbunden struktur ger modellen ett verklighetstroget resultat men tyg som har stora variationer i sin struktur visar sig vara svåra att beskriva med hjälp av modellen. Det sista problemet är att det tar väldigt lång tid att beräkna hur ljuset faller i en 3D-scen. Vi implementerar en renderare som använder strålföljning för att skapa realistiska bilder och som utnyttjar datorns grafikkort för att öka beräkningshastigheten till den grad att betraktningsvinkeln kan ändras interaktivt. Renderaren klarar av att återskapa objekt med komplex struktur och att simulera utseendet hos olika typer av material.

Acknowledgements

We would like to thank our supervisor, Marco Fratarcangeli, for helpful support and advice throughout the project. Thank you for giving us ideas and for all the inspiration.

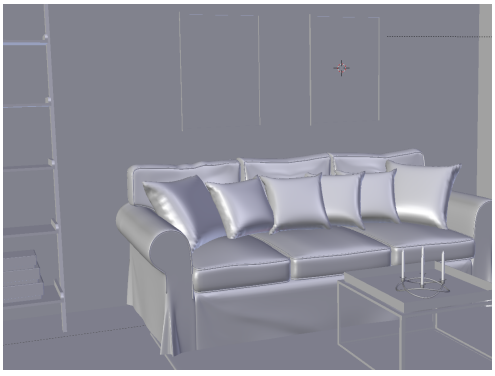
We would also like to thank Sebastian Ek and Anton Berg at IKEA Communications AB for the hospitality and the advice. We are grateful for the cloth samples you provided and for the guided tour of your offices.

Contents

Chapter 1

Introduction

For companies involved in selling physical goods, creating high-quality product images is often a necessity for communicating the range of available products to potential customers. Creating accurate and appealing product images is a craft which can involve a lot of work, especially when considering that many products often have multiple versions with subtle variations. Furthermore, when the images need to depict the products in a natural setting, there are many variables to take into account when considering international markets. For the setting to feel familiar it is, for instance, important to consider that the layout of a typical house might vary greatly between different climates and regions of the world.



(a) A preview of a scene with 3D modeled furniture, as seen in a 3D editor.

(b) The same scene after 3D rendering.

Figure 1.1: The rendering engine turns the geometrical description of the objects in *(a)* into the final image depicted in *(b)*. *Figure created by the authors. Unless otherwise specified, all following pictures, figures and illustrations are of the authors' creation.*

The amount of work involved in creating different variations of each product image quickly adds up, and producing the images in a traditional photo studio means transporting props and changing the stage and lighting for each permutation. This is a very labor-intensive process and there is an ongoing trend to instead produce product images through computer graphics. An example of an image created using this method is illustrated in Figure 1.1. Constructing a digital representation of the product is called *3D modeling*, and Figure 1.1a shows the 3D model of a room with furniture, as seen in a 3D editor. The resulting 3D model can be used to create realistic images through a process called *rendering*, the result of which is shown in Figure 1.1b. Since the products and surroundings are represented by digital 3D models, they are easy to modify and to replace. The process is also more labor and energy efficient, compared to building and photographing different scenes for all image variations. This means less environmental impact and also allows images to be produced at a higher rate.

This project has been done in cooperation with IKEA communications AB (ICOM), which is the company responsible for the IKEA catalogue. This means that they need to produce a very large number of product images each year. These images are currently created using both traditional photography and computer graphics. The latter method, however, is being used for an increasingly larger portion of the images. As described by Parkin (2014), more than 35 % of all product images were created using 3D graphics in 2012.

Producing images at a large scale requires a comprehensive workflow, where different parts of the image production team focus on separate tasks. Figure 1.2 shows a pipeline for image production using 3D graphics, which is divided into three main parts.

(a) 3D modeling. *(b)* Material definition. *(c)* Lighting and rendering.

Figure 1.2: The main stages of the content creation pipeline for 3D generated images used at a large company such as ICOM.

At the modeling stage, a 3D model of the geometry is built. Here, the model is often visualized as if it was made of gray clay, see Figure 1.2a. The reason for this is that it does not yet contain any information regarding the color of its surfaces.

In the *material definition* stage, all surfaces get assigned their color and other properties, such as the glossiness of the reflections and whether the object should be transparent, are defined, see Figure 1.2b.

During the final step, *rendering*, a renderer is used to calculate the way the light interacts with the models and materials in the scene. The renderer produces the final image, see Figure 1.2c.

While there are many benefits of generating images using computer graphics, there are also many technical challenges. Studying and developing solutions for these problems is a topic of active research in both academia and industry. In the context of a production environment, there are many areas in which the process could be improved.

1.1 Aim and Purpose

The aim of this project is to create tools and explore improvements that can alleviate some of the challenges in 3D content creation and rendering, as identified by ICOM. The work is focused on three specific problems related to the production of realistic images of furniture, each one related to a different stage of the 3D image production pipeline.

In the modeling stage, a tool for the 3D modeling of woven rattan furniture is developed. Creating 3D models of rattan furniture is time consuming and challenging and the tool automates part of this process by generating woven strands of rattan from simple user input.

In the material stage, studies are made of current shading models for representing woven fabrics. These are used as a base for developing a custom model which simulates the way light interacts with the individual threads in a piece of woven cloth. The aim is to develop a model which can create more realistic results than the method currently employed at ICOM, while requiring less input from the user. Evaluation of the model is done by comparing the results to samples of real cloth and to the current method used at ICOM.

Lastly, in the rendering stage, a renderer which utilizes the graphics processor for hardware acceleration is developed. The aim is to develop a renderer which can run at interactive frame rates, while still producing photorealistic images. The renderer should produce images at such a rate that there are no significant interruptions between changing a parameter and getting a visual result. In practice, this means producing images at a rate of at least two frames per second.

1.2 Report Structure

Since the project aims explore solutions in each of the three rendering stages, the report has been divided to into these into separate chapters. Chapter 2 acts as an introduction to the 3D-rendering pipeline. Chapter 3, 4 and 5 discusses the background, theory, and design choices more specific for each each explored solution. Performance statistics and visual appearances for the entire project is presented in Chapter 6, which are then discussed in more detail in Chapter 7.

Chapter 2

The 3D Image Production Pipeline

In large scale production, the process of creating photorealistic images using computer graphics is usually divided into smaller tasks. As mentioned in the introduction, a natural way to categorize these is to divide them into three main steps: *modeling*, where the geometrical structure of the objects is defined; *material definition*, where the color, texture and reflection properties of each surface is specified; and finally *rendering*, where a renderer is used to calculate the final image.

2.1 Step 1: Modeling

In the modeling stage, the geometry of all objects are defined as *triangle meshes*. A triangle mesh is a collection of triangles that combine to form the desired shape. Figure 2.1 shows an example of a triangle mesh, depicting a candle holder. Notice that the cylindrical shape of the candles is approximated by a number of flat triangles.

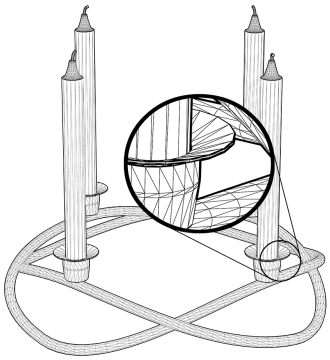
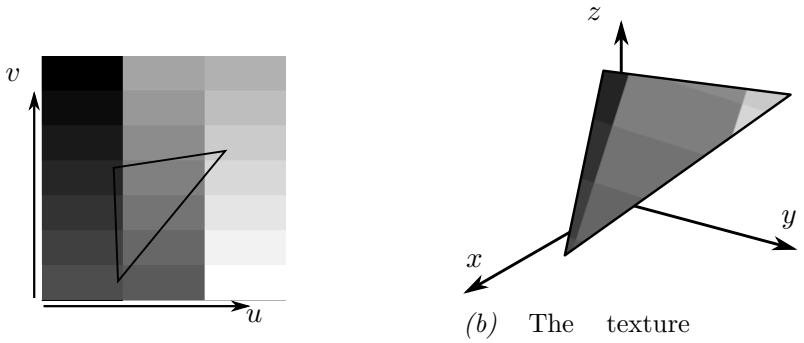


Figure 2.1: A triangle mesh in the form of a candle holder. Notice that the round shapes are approximated by several triangles.

Each triangle is defined by three *vertices*, which are points in 3D space. These points does often hold additional information, such as *texture coordinates*. Texture coordinates are 2D coordinates $(u, v) \in [0, 1] \times [0, 1]$ defined at each vertex. The texture coordinates are used when an image texture is to be applied to the mesh. An example of this would be when a pattern is applied to a 3D model of a cushion. The pattern would be described by an image and the texture coordinates define which part of the image should be mapped to each triangle. Figure 2.2 on the next page shows an example where a grayscale test image is mapped to a triangle.



(a) 2D texture coordinates.

(b) The texture mapped triangle in 3D space.

Figure 2.2: Texture coordinates are defined at each vertex of the triangle, describing how to map the image to it.

Besides triangle meshes, another useful way of defining 3D shapes is by using splines, which is a type of curve whose shape can be defined using a set of control points. In 3D modeling, splines are often used as input to tools which generate triangle meshes. A common use is to create a mesh by sweeping a profile along a path, see Figure 2.3.

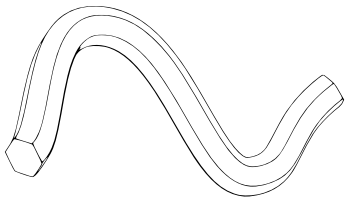


Figure 2.3: A 3D model created by sweeping a hexagonal profile along a path.

2.2 Step 2: Material Definition

The material definition step consists of specifying the visual characteristics of all surfaces in the scene. In this step, different *materials* are created and applied to the 3D models. A material determines the color of a surface, but also how reflective it is, whether the object is transparent, and many other properties. The behaviour of a material is defined by a program called a *shader*. The shader will be called by the renderer whenever it needs to know the amount of light that reflects off a surface.

(a) A sphere showcasing diffuse reflection.

(b) A sphere with both diffuse and specular reflection.

Figure 2.4: Examples of two materials where one is exhibiting exclusively diffuse reflection and the other a combination of both diffuse and specular reflection.

It is common to separate the reflection of light into two components, diffuse and specular reflection, see Figure 2.4. The specular reflection is described by the Fresnel equations,

where a portion of the light is reflected off the surface and the rest is transmitted through it. Diffuse reflection, on the other hand, is the result of light penetrating the surface and being scattered in random directions from inside the medium (Hoffman, 2013).

Often, images are used to vary the color of a material across a surface. This is an example of using a texture to influence a parameter of the material. Textures can be used to control different aspects of the material. For instance, the paint on a metal object might be chipped, with some of the metal showing through. Such a material could be defined by using a texture which makes the surface more reflective at the areas where the metal is visible.

Furthermore, a texture need not necessarily be defined by an image. The variation could also be determined by a mathematical function that assigns a color to each (u, v) texture coordinate. For example, a classic checkerboard pattern on a surface could easily be produced using a mathematical function, thereby circumventing the need to create an image and storing it in memory.

2.3 Step 3: Rendering

Rendering is the part of the pipeline where the final image is computed. A renderer is used to compute the way light interacts with the scene and how it is reflected at the surfaces. The two most common methods for rendering are *rasterization* and *ray tracing*. While both methods are widely used, ray tracing is usually the preferred method for photorealistic image production, since it is capable of producing very good images with realistic lightning. Rasterization, on the other hand, is usually employed in games and other applications that need to run in realtime. The reason for this is that rasterization is generally faster and less computationally intensive than ray tracing.

Figure 2.5: An example of a scene rendered with ray tracing. Notice the realistic reflections in the glass and the metal.

Ray tracing involves sending rays into the scene and finding their intersection with its geometry. At each intersection, a shader is called upon to calculate the color of the ray. The shader may send out additional rays, in order to simulate reflected or refracted light. This gives the ray tracing algorithm a recursive behaviour and allows for a realistic simulation of light. It is also the reason why ray tracing is often slower than rasterization, since it results in more work and a much less coherent memory access pattern (Akeley, Kirk, Seiler, Slusallek, & Grantham, 2002). Figure 2.5 shows an example of a scene rendered using ray tracing, notice the realistic reflections and the refraction of the root of the acorn, which would be difficult to recreate using rasterization.

Chapter 3

Modeling – Rattan Furniture

This chapter will explore the modeling stage of the 3D image production pipeline, and how procedural tools can help artists in their work. In this stage, the 3D shape of the objects are defined and therefore the focus is on the geometry of rattan furniture. A rattan chair is made out of a rigid frame interwoven with flexible pieces of rattan wood. Because of its complicated geometrical structure, it is very time consuming to create detailed 3D models of these types of furniture. The most important goal of the project is the visual quality of the results. The aim is to generate detailed and convincing 3D models that can be used in the rendering of photorealistic images.

Figure 3.1: A photo of a rattan chair. "Rattan Chair" by Chris 73 (2005) licensed under CC BY-SA 3.0 .

3.1 Background

Rattan furniture is made out of long wooden strands which are woven and fixed to a rigid frame which gives the furniture its complicated appearance. An example of a rattan chair can be seen in Figure 3.1. In 3D graphics, there are multiple approaches that may be used to describe the geometry of these kinds of detailed 3D shapes. The most direct approach is to simply model the object by hand at high resolution and making sure to include enough detail. Depending on the object, this can be a large undertaking. Rattan furniture, with all of its interwoven features, is an example of an object which would be labor-intensive

to model by hand. Other approaches, often utilized for fine, repeatable details, is to use a texture to alter the way light is simulated during rendering, so called *bump-mapping*. Another use of textures is to displace the vertices of the mesh, so called *displacement-mapping*. These techniques allow artists to make a simple 3D model but still give it the appearance of having finer detail. There are limitations to these techniques, however, and the interwoven nature of rattan furniture are not well suited to these techniques.

The current approach for generating 3D models of rattan furniture at ICOM is to use 3D scanning to capture the geometry. This is one of the few types of objects that are considered to be too complicated to model by hand. Unfortunately, the quality of the scanned models are not high enough to allow them to be used in the foreground of images. Instead, rattan furniture is restricted to be used as background elements.

The modeling of rattan chairs proves to be a problem. A tool which can aid the artist during the modeling process could help alleviate much of the work involved in modeling rattan furniture. In the rest of this section, some theory behind splines will be presented. These are the fundamental building blocks of the tool for generating rattan furniture which will be developed in this chapter.

3.1.1 Splines

As described in Section 2.1, splines are an important tool in 3D creation. This section describes the mathematical foundations of the two types of splines that were used in the rattan tool, *Bézier curves* and *NURBS*.

Bézier Curves

A general Bézier curve of degree n is defined by the control points $\boldsymbol{P_0}, \ldots, \boldsymbol{P_n}$ and can be expressed by the sum

$$\boldsymbol{B(t)} = \sum_{i=0}^{n} b_{n,i}\boldsymbol{P_i}, \qquad t \in [0,1], \tag{3.1}$$

where $b_{n,i}$ are the *Bernstein basis polynomials*, which are defined as

$$b_{n,i} = \binom{n}{i} t^i (1-t)^{n-1}$$

(Prautzsch, Boehm, & Paluszny, 2002). In computer graphics, it is common to use Bézier curves of degree 3. Having a degree of 3 gives enough control over the shape of the curve, while still keeping the degree of freedom at a reasonable level.

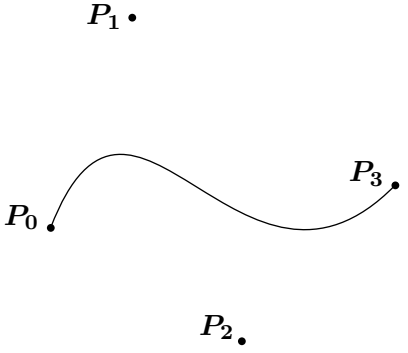


Figure 3.2: An example of a Bézier curve of degree 3. Notice that the curve passes through its first and last point, but that the other two points do not lie on the curve.

A Bézier curve has the property that it starts at the first control point, $\boldsymbol{P_0}$ and ends at the last one, $\boldsymbol{P_n}$. For a degree 3 curve, the remaining points, $\boldsymbol{P_1}$, $\boldsymbol{P_2}$, are used to define

the shape of the curve, as can be seen in Figure 3.2. When a longer or more complex curve is needed, it is common to piece together several Bézier curve segments end to end. This creates a shape which is easier to work with, than if a single, high degree curve would have been used. All curves used in this project are of degree 3.

NURBS

The *Non-Uniform Rational B-Splines* (NURBS) described by Piegl and Tiller (1997) are a type of curves which are defined similarly to Bézier curves, but with a different set of basis functions, which are more complex than the Bernstein basis functions. While a Bézier curve is fully described by its control points, in a NURBS curve each control point also has a weight which determines how much each point influences the shape of the curve.

A NURBS curve makes it easy to control its continuity, or ”smoothness”. In certain contexts it is important to use curves and surfaces with a certain level of continuity. This is one reason why NURBS are widely used in 3D graphics.

3.2 Related Work

There are works that cover the creation of detailed cloth geometry, but apart from the similar woven nature between the two materials, the techniques are not applicable to the geometry of rattan furniture. Furthermore, while many approaches to rendering cloth use textures, rattan furniture needs detailed geometry in order for the model to look realistic. Rattan furniture has gaps between the strands that let light pass through and give rise to shadow patterns. The large geometry has a significant impact on how light interacts with the material.

There has been some interesting work on generating 3D models of knitted fabrics, notably that by Kaldor, James, and Marschner (2008), which was later developed by Cirio, Lopez-Moreno, Miraut, and Otaduy (2014). This work used splines together with a sophisticated physics model based on Lagrangian mechanics.

Little other work on generating such geometries has been found closely related to the problem of generating rattan furniture. An inspiration, however, has been found in websites that sell 3D models of rattan chairs.

3.3 Restrictions

Automating the generation of rattan geometry is a problem and there are many interesting advances that could be made. The aim of this project has been to create a tool which could help in the 3D modeling of a simple rattan chair.

Most rattan strands have a diameter of a few millimeters. When decreasing the size of the strands, benefits of explicitly describing the geometry are lost. Smaller strands gives a more dense pattern which in turn means a model that takes longer to render. This project will only focus on weaved geometry with a strand size of a few millimeters.

More advanced pieces of rattan furniture use different weaving patterns in order to join together different sections or to give added detail. This project will only focus on one plain weaving pattern.

The tool will not consider the organic variations present. This is a potential drawback since strands used for rattan furniture are rarely completely uniform.

In the creation of a realistic rendering of a rattan chair, both the 3D model and the material definitions of the object are important for a good end result. This project will focus on modeling; exploring textures and materials is not considered.

3.4 Method

This section describes the process, design choices and implementation of an add-on for the 3D package Blender which generates a curved section of woven rattan.

The first step was to create a simple flat section of woven rattan. For this, *NURBS* were used. Control points were placed such that the shape of the splines resembled woven strands of rattan. The second step was to give the strands the ability to wrap around a frame. Finally, in order to achieve curved rattan surfaces, the flat woven plane was transformed into 3D space. By transforming the control points of the NURBS, the shape could be manipulated in such a way that the rattan geometry could be adapted to fit the structure of a chair.

Since the goal of this project was to create a prototype of a tool that could be used in production, care was taken to only use standard and well understood features available in most 3D modeling applications. The reasoning behind this was to ensure that the tool could be ported to different 3D modeling packages in the future.

Many 3D packages allow custom tools to be defined, using either a general purpose programming language or a domain specific language. Since Blender uses Python for scripting, the tool was developed using this language.

3.4.1 Weaving Rattan Strands

A woven rectangular section of rattan was created using NURBS. The reason for using NURBS instead of Bézier curves is that while Bezier curves makes it easier to position the strands exactly, NURBS makes it easier to control the curvature of the shape.

To create the woven geometry, the tool creates a NURBS with control vertices placed at regular intervals. The control points of the NURBS are transformed into an oscillating wave shape. The control vertices in each spline are shifted so that they interweave with each other. Another set of splines are placed perpendicularly to these in order to generate the straight pieces of rattan which are used as a base for the weave.

The weight parameter of the NURBS spline control points is set to a low value, around 0.2. This results in a curve with a low curvature even with few control points. This makes the implementation easier since it is possible to create a good looking weave from a sparse grid of control points.

The splines are given a uniform thickness by sweeping a circular profile along them. This is done by using the `bevel depth` parameter in Blender.

3.4.2 Creating Curved Surfaces

The algorithm for generating interwoven strands of rattan described in the previous section produces a pattern on a plane. In order to create curved surfaces, the geometry needs to be transformed. This transformation is defined by two Bézier curves, an example can be seen in Figure 3.3 shown on the following page.

Bézier curves were chosen since their simple definition makes it possible to match the curve shown in the interface exactly, using the formulas in Section 3.1.1. Bézier curves are commonly used in 3D packages and are familiar to most users. Using two Bézier curves allows the user to define a range of a surface shapes for the pattern.

The other input to the tool is the radius of the strands. This is the radius of the profile shape which is swept along the strands, as described in the previous section.

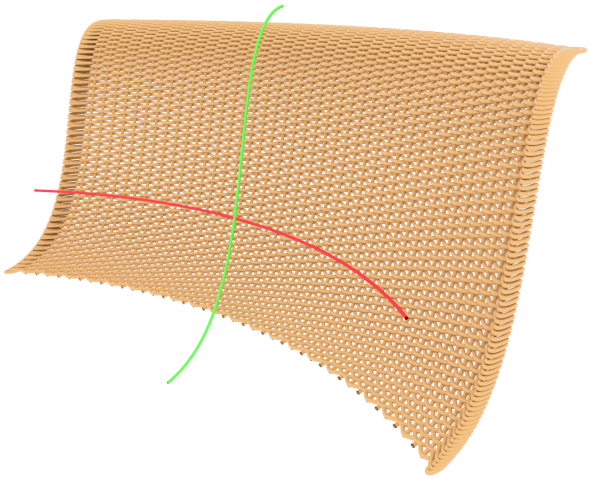


Figure 3.3: An rendition of two Bezier curves used for input to the tool. The corresponding output is seen behind.

3.4.3 Spacing of Strands

The strands that are used in rattan furniture tend to be packed close together but do not intersect. In order to achieve this in the generated geometry, the number of rows and columns of NURBS to use for the interwoven pattern has to be calculated. The number is calculated from the length of the two input curves, as introduced in the previous section together with the strand thickness.

The number of strands was calculated as

$$\text{Number of strands} = \frac{\text{Length of input curve}}{p \cdot \text{Thickness of strand}} \tag{3.2}$$

where the parameter p was varied in order to get a good result.

3.4.4 Connections With the Frame

Special care had to be taken regarding the way the rattan strands connect to the frame. Rattan chairs are often constructed from a few long strands which span multiple rows. In the rattan chairs that were examined, the strands connected to the frame by turning around the frame structure.

The tool handles these connections to the frame by extending the splines making up the pattern and twisting them around a cylinder of fixed radius. The turn is done in such a way that the strands twist and join into the next row of NURBS in the interwoven pattern.

3.5 Chapter Summary

In this chapter, the process, design choices and implementation of a tool for rattan furniture have been presented. The tool took the form of an add-on for the 3D package Blender which generates a set of splines that have the appearance of woven rattan.

It was described why these types of objects are a challenge in 3D rendering of furniture and why rattan furniture is a problem in the production of photorealistic product images today.

Lastly, the approach taken to find improvements to this problem and the challenges that were faced during the design and implementation of the solution were discussed.

Chapter 4

Material Definition – Cloth Shader

In the 3D content creation pipeline, the shader is meant to be applied at the material definition stage. A shader is a small program which is called during rendering each time a ray intersects a surface. The shader implements a mathematical model which returns the amount of light that is reflected in a specific direction. This chapter is devoted to the development of a shader that simulates the appearance of woven cloth.

From studying existing and well-regarded shading models for woven cloth, the hope was to learn the theory and methods used to achieve realistic results. One of these models was then selected as a base for a custom implementation. The goal of the project was to create an implementation that could be used in production at ICOM. This means implementing the shader as a plug-in for the tools used there, the renderer V-Ray (Chaos Software, n.d.) and the 3D authoring software Autodesk 3ds Max (Autodesk, n.d.).

4.1 Background

The common method of simulating the appearance of woven cloth is to use detailed textures together with a general purpose shader. In conversations with ICOM, it was stated that this is the method that they currently employ.

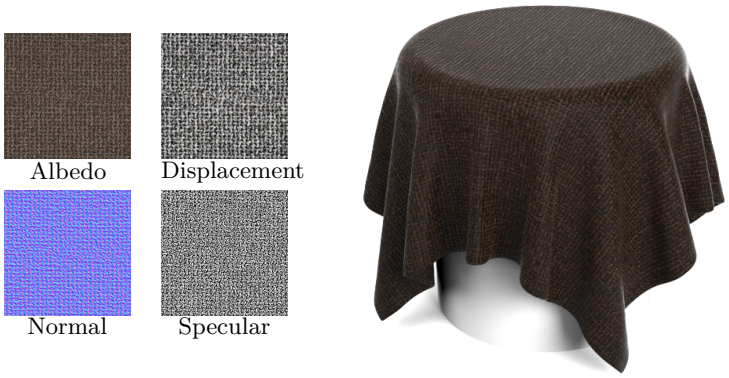


Figure 4.1: An example of using textures as input to a general purpose shader in order to simulate the appearance of woven cloth. The large image shows a mesh with the general purpose shader applied. Notice the subtle bumps of the yarn.

An example of this method can be seen in Figure 4.1, where all textures for a certain fabric are shown, together with the resulting image. The Albedo texture informs the shader of the coloring of the surface, while normal, displacement and specular textures govern other effects.

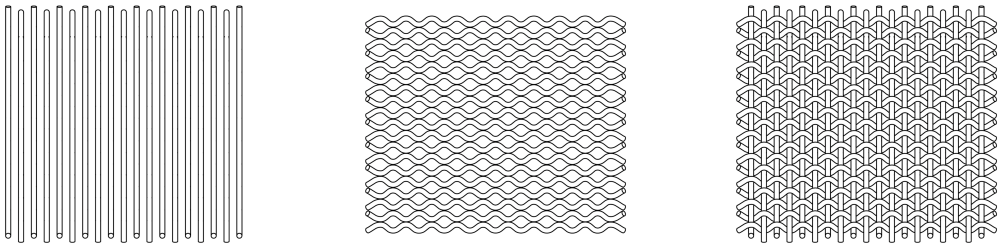
Since each individual texture can be modified, this method gives an artist rich control over the look of the material. Even so, this approach has its drawbacks. New textures have to be created for each new type of fabric, which is very labor-intensive. Another drawback is that these textures often needs to have a high resolution and therefore consume a considerable amount of memory. Since the material is created using a general-purpose shader, some visual effects can not be correctly simulated. A good example would be the appearance of satin, which is very difficult to reproduce using this technique.

There are other approaches for simulating the appearance of woven cloth. One is to explicitly describe the geometry of the individual yarns in high detail, giving impressive results but quickly becoming computationally expensive when considering larger pieces of cloth. Another method, which is the one this chapter presents, is the development of a custom shader specifically for woven cloth.

In the remainder of this background section, some fundamental concepts of woven cloth are introduced, as well as shading-related terms and concepts. Lastly, prior work in the area of shading models for woven cloth are discussed.

4.1.1 Fundamentals of Woven Fabric

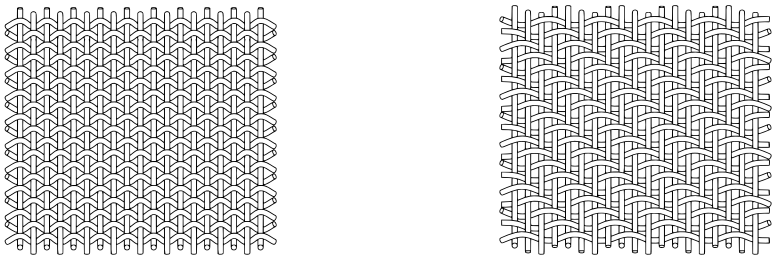
The basic process of weaving consists of interlacing yarns in different patterns using a loom. Two sets of yarn are used, the *weft* runs along the loom and the *warp* runs orthogonal to it, see Figure 4.2. The loom lifts a selection of the warp threads, allowing the weft to be slid through, passing below the raised warp threads and above the others. This process is repeated until the fabric is complete (Adanur, 2001).



(a) Warp. *(b)* Weft. *(c)* Complete weave.

Figure 4.2: The warp and weft in a woven fabric. The weft runs parallel to the loom and the warp runs orthogonal to it.

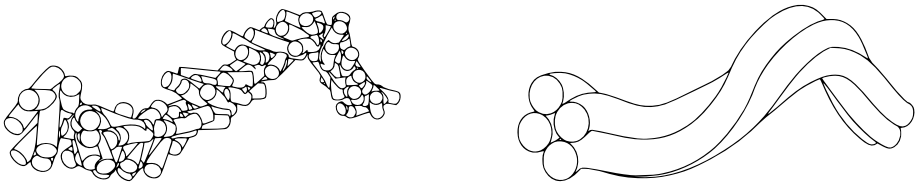
The sequence in which the warp and weft threads are interwoven is defined by a *weaving pattern*. Fabrics which are woven with different patterns have different physical and visual characteristics, even if the same type of yarn is used. Figure 4.3 shows two examples of common weaving patterns: plain weave and twill (Sew4Home, 2010).



(a) Plain Weave. *(b)* Twill.

Figure 4.3: Two examples of common weaving patterns. Plain weave, *(a)* is perhaps the simplest possible weaving pattern and twill, *(b)* is the pattern used in, for instance, denim clothing.

Weaving can be done using various types of yarn. Most of these can be divided into two categories. *Staple* yarn is made from materials such as cotton, which consists of several short filaments that are spun together and *filament* yarn, which consists of long filaments which run side by side along the thread. See Figure 4.4 for an illustration of the two types. Examples of filament yarn include polyester and silk. The more smooth and



(a) Staple yarn. *(b)* Filament yarn.

Figure 4.4: The two most common types of yarn. Staple yarn, such as cotton, consists of short filaments twisted together. Filament yarn consists of long, continuous filaments.

coherent structure of filament yarn means that it often exhibits a more focused highlight compared to staple yarn.

4.1.2 The Bidirectional Reflection Distribution Function

Implementing a shading model, which is the purpose of this chapter, is synonymous with defining a Bidirectional Reflection Distribution Function, or BRDF, $f_r(\boldsymbol{x}, \vec{\omega_i}, \vec{\omega_o})$. This function describes the amount of light which is reflected from the direction $\vec{\omega_i}$ to the direction $\vec{\omega_o}$ at a point $\boldsymbol{x}$ on a surface. Perhaps the simplest example of a BRDF is the *Lambert* BRDF, which describes an ideal matte surface. Such a surface will reflect light evenly in all directions. Because each ray of light will have an equal probability of being reflected in any direction, the Lambert BRDF independent of both $\vec{\omega_i}$ and $\vec{\omega_o}$, and is described by the constant expression $f_{Lambert} = \frac{1}{\pi}$ (Pharr & Humphreys, 2010).

In order to be well-behaved and physically plausible, a BRDF needs to satisfy some conditions, namely that for any outgoing direction $\vec{\omega_o}$ and surface position $\boldsymbol{x}$, three properties should be fulfilled (Pharr & Humphreys, 2010).

The first property, *Helmholtz reciprocity*,

$$f_r(\boldsymbol{x}, \vec{\omega_i}, \vec{\omega_o}) = f_r(\boldsymbol{x}, \vec{\omega_o}, \vec{\omega_i}), \tag{4.1}$$

describes the fact that the result of the BRDF should be the same even if the outgoing and incident directions are exchanged (Iwanaga, Hatano, Ishihara, & Vengurlekar, 2006).

The second property, *Positivity*,

$$f_r(\boldsymbol{x}, \vec{\omega_i}, \vec{\omega_o}) \geq 0, \tag{4.2}$$

simply means that the BRDF cannot take negative values, which is a fundamental property of light, a surface cannot reflect less than nothing.

The final property, *Energy conservation*,

$$\int_\Omega f_r(\boldsymbol{x}, \vec{\omega_i}, \vec{\omega_o})(\vec{\omega_i} \cdot \vec{n})\mathrm{d}\vec{\omega_i} \leq 1, \tag{4.3}$$

ensures that a surface cannot reflect more light than that which arrives at it. Ω is the hemisphere of directions above the surface at the point $\boldsymbol{x}$, $\vec{n}$ is the normal of the surface and the factor $(\vec{\omega_i} \cdot \vec{n})$ describes the weakening effect which occurs when the arriving light is not perpendicular to the surface.

4.2 Related Work

Interesting research related to the visual appearance of fabrics include the work by Yuksel, Kaldor, James, and Marschner (2012), which present a method for generating geometry for knitwear. The tension of the yarn is simulated in a pre-processing step to move the geometry into its natural rest state. Another approach is presented by Zhao, Jakob, Marschner, and Bala (2012), which uses volumetric data from micro-CT scans of real fabrics. Both of these approaches represent the geometry of a fabric at the yarn level.

The lecture notes by Schröder, Zhao, and Zinke (2012) provides a good overview over the state of the art in physically based rendering of cloth, as of 2012. More recent works, such as that by Khungurn, Schroeder, Zhao, Bala, and Marschner (2015) discuss matching parameters for *micro-appearance* models of cloth to real world data. Although this work is not immediately related to the aims of this project, it does provide an overview of the currently available models for cloth appearance.

For the purpose of this project, the work by Irawan and Marschner (2012), and that by Adabala, Magnenat-Thalmann, and Fei (2003), are probably the most relevant. These models use a mathematical model for the light interaction with the cloth and only need simple pattern data as input. Further developments to this work are given by Yang (2009), who describes additions that allows the model to handle self-shadowing and masking.

Furthermore, materials related to the BRDF includes that of Heitz (2014), which gives an introduction to shadow and masking functions, and the work by Wang, Xie, and Krishnamachari (2014), which describes how to implement importance sampling for cloth shaders. The course notes by Hoffman (2013) give a good foundation to the theory and process of creating physically based BRDF models, and works such as that by Hanrahan and Krueger (1993) and lecture notes by S. Marschner (2007) give an introduction to the theory behind volumetric scattering.

4.3 Restrictions

As the aim of this project is to create a shader that can give a surface the appearance of woven cloth, textiles created through other methods, such as knitting or stitching, are not considered.

When weaving a real fabric using a loom, the yarn used for each warp and weft thread can be chosen independently. This means that a piece of woven cloth can consist of several different types of yarns. This freedom is sometimes used in order to give the cloth certain characteristics. The model will not support any parameter except for the color to specified individually for each thread.

The shader is meant to be used in scenes where the object is at some distance from the camera. No attempt will be made to make the cloth look realistic in closeup pictures. Effects such as translucency, displacement, self-shadowing and fuzziness stemming from stray filaments are less important from a distance and are therefore not considered.

4.4 Method

The model used in the shader is based on the work by Irawan and Marschner (2012) and represents the yarn as torus segments, or bent cylinders. Conceptually, these yarn cylinders consist of individual filaments that are twisted around the surface. The shader implementation needs to evaluate where to place the yarn cylinders and does this by reading and interpreting weaving patterns. During rendering, the interaction between the light and the yarn cylinders are calculated based on a sophisticated reflection model. In

order to improve realism, different types of noise are added to the shader in order to emulate the natural variation often present in real fabrics.

In the remainder of this section, these key components are presented and discussed in detail.

4.4.1 Reading and Interpreting Weaving Patterns

One of the most defining features of a woven fabric is its weaving pattern. It is therefore important that the pattern used by the shader can be easily changed. There is a standard file format for weaving patterns, called *WIF*, or *Weaving Information File*, which is a text based format supported by most weaving software (Nielsen, 1997). The information from a WIF file can be parsed into a matrix where each entry represents an intersection of warp and weft, and contains a value indicating which thread is on top of the other. It also includes the color of the topmost yarn.

Since the data in a WIF file is presented in a way that is relevant for weaving, it is not necessarily ideal for rendering. For this reason, a simpler file format, called *weave* files, was created during the development of the cloth shader. The purpose of this file format was to be able to input custom patterns without creating a complete WIF file. A weave file contains the color and size of the warp and weft threads, and a matrix which describes the weaving pattern. Since most weaving software uses WIF files, the cloth shader was designed to allow both WIF files and weave files.

4.4.2 Representing the Yarn Geometry

From reading one of the two types of pattern files, as discussed in the previous section, a matrix representing each intersection of the pattern is created. During rendering, the shader is called with a set of the texture coordinates given by the renderer, here called (u_t, v_t). These coordinates will later transform into different coordinate systems that facilitate the description of the geometry and evaluation of the model.

In a weave pattern, each thread might pass over several orthogonal threads at a time. Each such continuous run of thread was treated as a single *yarn segment*. A yarn segment starts and ends when it is being covered by another thread.

A new *yarn local coordinate system* is introduced, in which coordinates are denoted by (x, y) and are defined in such a way that y always runs along the length of the yarn and x runs across it. Figure 4.5 shows an example of such a segment, where the weft passes over two warp thread as well as an illustration of the yarn local coordinate system. The yarn local coordinates are in the range $x, y \in [-1, 1]$, where $(x, y) = (-1, 1)$ would be the top left of a yarn segment and $(x, y) = (1, -1)$ would be the bottom right.

In real cloth, a visible yarn segment will have hidden yarns running perpendicularly under it. This gives the yarn segment a bent shape. To approximate this behaviour, the yarn segments are treated as bent cylinders. A coordinate transformation is derived by considering the angles and relations involved in representing a coordinate on a bent cylinder. The yarn local coordinates (x, y) can be transformed to the surface of such a cylinder using the mapping

$$u = \arcsin\left(2\frac{y}{l}\sin(u_{max})\right)$$
$$v = \arcsin\left(2\frac{x}{w}\right),$$

where l and w are the length and width of a yarn segment, respectively. The values for l and w were calculated during the first transformation to the yarn local coordinates.

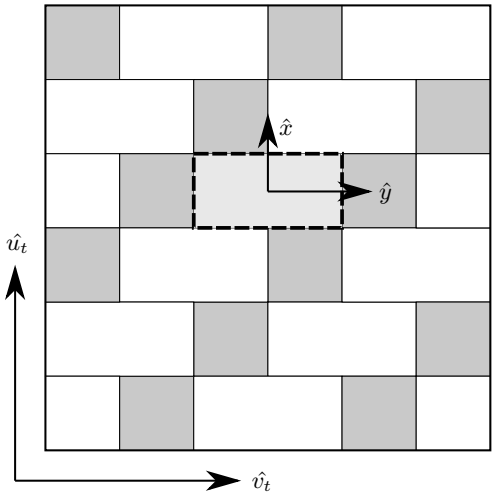
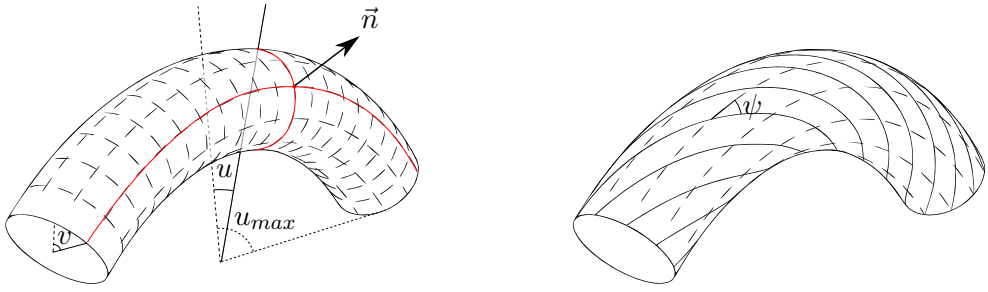


Figure 4.5: A pattern matrix with a highlighted yarn segment. The corresponding yarn local coordinate system is also illustrated. The y-axis is defined such that it always runs along the yarn, while the x-axis runs across it.

Figure 4.6a illustrates the roles of the newly mapped coordinates (u, v). The coordinate u is the angle from the center of the yarn segment to the current point along the cylinder and is in the range of $[-u_{max}, u_{max}]$, where u_{max} is a parameter which describes how much the cylinder should be bent. The v coordinate is the angle from the center of the segment to the current point across the cylinder and is in the range $[-\pi, \pi]$. The normal of a point (u, v) on the surface of the cylinder is given by

$$\vec{n}(u, v) = \begin{bmatrix} \sin v \\ \sin u \cos v \\ \cos u \cos v \end{bmatrix}.$$

Another parameter, ψ, determines the *twist* of the yarn. This describes the angle the filaments make with the yarn cylinder, see Figure 4.6b. While staple yarn often exhibits strong twisting, this effect is not very pronounced in filament yarn, thus $\psi_{filament} \approx 0$. The tangent vector, $\vec{t}$, is defined to be orthogonal to the normal and parallel to the direction of the filaments.



(a) *(b)*

Figure 4.6: (a) The coordinate u is the angle to the current position along the length of the yarn and the coordinate v is the angle along the short side of the yarn. The vector $\vec{n}$ is the normal of the bent cylinder surface. *(b)* The twist parameter ψ determines the angle the filaments make with the yarn.

4.4.3 Calculating Reflection from the Yarn

A yarn is made up of many smaller filaments that are packed together. These filaments are often somewhat translucent, and once the light enters the yarn, it is scattered by the reflection from its constituent filaments. Some of the light will continue on its way, but at each intersection with a filament, part of the light will be reflected. For simplicity, the model does not calculate the intersection with each individual filament. Instead, it considers the average number of intersections that would occur when the light travels a certain distance in the yarn. It is assumed that when the light intersects a filament, some of it is reflected specularly, and some of it is transmitted through the filament. Furthermore, a portion of the transmitted light will be absorbed by the filament, which means that less light will leave the filament than that which entered it. When all these factors are considered, the resulting BRDF is the product of three factors,

$$f_r(\boldsymbol{x}, \vec{\omega_i}, \vec{\omega_o}) = \underbrace{f_p(\vec{\omega_i}, \vec{\omega_o})}_{\textit{Phase function}} \overbrace{A(\boldsymbol{x}, \vec{\omega_i}, \vec{\omega_o})}^{\textit{Attenuation function}} \underbrace{G(\boldsymbol{x}, \vec{\omega_i}, \vec{\omega_o})}_{\textit{Geometry factor}}, \tag{4.4}$$

where $\boldsymbol{x}$ is the current shading point, and $\vec{\omega_i}$ and $\vec{\omega_o}$ are the incident and outgoing directions of light, respectively.

The *phase function*, f_p; the *attenuation function*, A; and the *geometry factor*, G, represent the different factors that affects the light. The phase function describes the scattering of the light due to intersection with the filaments in the yarn. The attenuation function describes the part of the light that is absorbed by the yarn. The final component of the equation, the geometry factor, does not describe any particular physical phenomenon. Instead, it is the result of transformations between coordinate systems. In the shader implementation the three factors are defined as in the work by Irawan and Marschner (2012). The following three sections will discuss the three factors.

The Phase Function

The phase function, f_p, describes the scattering inside the yarn and simulates the way light interacts with the filament fibers. It determines the amount of light which is scattered from the direction $\vec{\omega_i}$ to the direction $\vec{\omega_o}$ at any point. However, f_p does not depend on the exact values of $\vec{\omega_i}$ and $\vec{\omega_o}$, but only on the angle between them.

Irawan and Marschner (2012) introduce two new parameter variables α and β and use the phase function

$$f_p(\vec{\omega_i}, \vec{\omega_o}, \alpha, \beta) = \alpha + g(\vec{\omega_i} \cdot \vec{\omega_o}, 0, \beta),$$

which is the sum of a constant term, α, and the *von Mises* probability distribution $g(\cos\theta, a, b)$, which is analogous to the normal distribution but defined in terms of an angle, θ. a corresponds to the mean parameter of the normal distribution, and describes which angle the values will be centered around. The parameter b describes the concentration of values around a, and corresponds to the inverse of the variance in the normal distribution. When used in the phase function, the mean parameter is set to $a = 0$, which means that the scattered light will be evenly distributed around the same direction it was originally traveling. The concentration parameter is set to $b = \beta$, which gives the user control over the amount of scattering of the light. Together, the parameters α and β of the phase function factor can be used to change the behaviour of how the light is scattered in the yarn.

The Attenuation Function

The attenuation function, $A(\boldsymbol{x}, \vec{\omega_i}, \vec{\omega_o})$, is used to model the fact that some of the light is absorbed by the filaments of the yarn. Irawan and Marschner (2012) take advantage

of *Seeliger's law*, which describes the absorption inside a homogeneous medium under a flat surface. The assumption of a flat surface is not true of a yarn cylinder, but the error introduced by this approximation is minimal. Seeliger's law is given by

$$L_o(\vec{v_i}, \vec{v_o}) = \frac{\vec{v_i} \cdot \vec{n}}{\vec{v_i} \cdot \vec{n} + \vec{v_o} \cdot \vec{n}} L_i(\vec{v_i}, \vec{v_o}),$$

where $\vec{n}$ is the normal of the surface, L_i is the incident light, L_o is the outgoing light, and the directions $\vec{v_i}$ and $\vec{v_o}$ are ingoing and outgoing directions, respectively (Hanrahan & Krueger, 1993). Figure 4.7 illustrates the geometry involved when light intersects with a yarn cylinder.

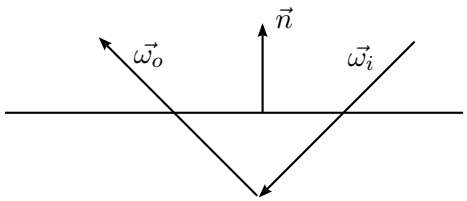


Figure 4.7: The incident and outgoing directions of a ray being scattered by the yarn.

Seeliger's law needs to be applied both to the incident and outgoing rays, since the light is attenuated equally both on its way into, and out of, the yarn. Combining both paths results in the aggregated attenuation function

$$A(\boldsymbol{x}, \vec{\omega_i}, \vec{\omega_o}) = \frac{(\vec{\omega_i} \cdot \vec{n})(\vec{\omega_o} \cdot \vec{n})}{\vec{\omega_i} \cdot \vec{n} + \vec{\omega_o} \cdot \vec{n}}.$$

The Geometry Factor

As mentioned above, the geometry factor, G, does not model any specific phenomenon. Instead, it is the result of the parametrization of the yarn cylinders. It can be found by considering an integral over an entire yarn segment, as shown by Irawan (2008, page 64). However, the derivation is quite involved and not relevant to the current discussion. For staple yarn and setting $\boldsymbol{x} = (u, v)$, the geometry factor is

$$G(u, v, \vec{\omega_i}, \vec{\omega_o}) = \frac{a(R(u) + a\cos v)}{(\vec{n} \cdot (\vec{\omega_i} + \vec{\omega_o}))\sin\psi}, \tag{4.5}$$

and for filament yarn

$$G(u, v, \vec{\omega_i}, \vec{\omega_o}) = \frac{a(R(u) + a\cos v)}{\left|\left(\vec{t} \times (\vec{\omega_i} + \vec{\omega_o})\right)_x\right|}, \tag{4.6}$$

where a is the radius of the yarn cylinder, u and v are the coordinates of the yarn segment, ψ is the twist angle, and $\vec{t}$ is the tangent vector. These coordinates were all introduced in Section 4.4.2. $R(u) = (\sin u_{max})^{-1}$ is the radius of curvature of the cylinder, and finally, $(\vec{v})_x$ represents the magintude of $\vec{v}$ along the x coordinate of the yarn cylinder.

4.4.4 Finding the Location of Specular Reflection

According to the well known *Law of Reflection*, ideal specular reflection occurs only when the angle of incidence is equal to the angle of reflection. Or equivalently, when the half vector $\vec{h} = (\vec{\omega_i} + \vec{\omega_o})/||\vec{\omega_i} + \vec{\omega_o}||$ is equal to the normal (Blinn, 1977). However, in the case of the model presented by Irawan and Marschner (2012), each filament is modeled by an infinitesimally thin cylinder oriented along the tangent vector $\vec{t}$. The normal is assumed

to point outwards, which means that the condition $\vec{h} = \vec{n}$ is equivalent to the condition that $\vec{h}$ is perpendicular to $\vec{t}$, i.e. that

$$\vec{h} \cdot \vec{t} = 0. \tag{4.7}$$

If $\vec{\omega_o}$ and $\vec{\omega_i}$ are kept constant, this condition will only be true for an infinitesimally thin band of locations across the yarn cylinder. This imposes a restriction on the u, v coordinates, which are thus functions of one another. The shape of the band will change depending on the twist parameter, ψ, introduced in Section 4.4.2, which controls the direction of the tangent vector.

A problem with the model, as described so far, is the fact that the highlight is infinitesimally thin. This means that it would be impossible for a ray to hit it during rendering. Irawan and Marschner (2012) propose to widen it, in an ad-hoc emulation of glossy reflection, into a band of constant width. This is done by considering any shading point which is within Δx of the highlight to also be part of it. The result is that the highlight is expanded into a band of constant width, instead of being infinitesimally thin.

4.4.5 Normalizing the Reflection

The condition in Equation (4.3) states that the integral of the BRDF over the hemisphere should be less than or equal to 1. In order to ensure this, a normalization factor

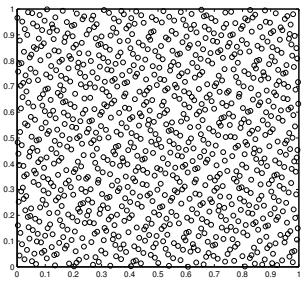
$$a = \left(\max_{\boldsymbol{x}, \vec{\omega_i}} \int_\Omega f_r(\boldsymbol{x}, \vec{\omega_i}, \vec{\omega_o})(\vec{\omega_i} \cdot \vec{n}) \mathrm{d}\vec{\omega_o} \right)^{-1} \tag{4.8}$$

was introduced. The maximization has to be done for all points $\boldsymbol{x} = (u, v)$ such that $u \in [0, 1]$ and $v \in [0, 1]$ to make sure that the shader is normalized over the entire piece of cloth, and for all values of $\vec{\omega_i} \in \Omega$, where Ω is the hemisphere of directions above the surface, to make sure it is normalized for all incident angles. By normalizing the BRDF with the maximum value over these dimensions, the condition in Equation (4.3) is fulfilled.

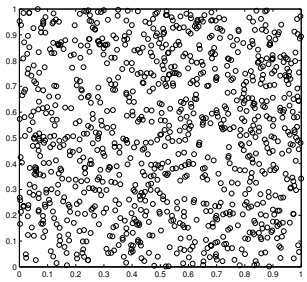
The normalization factor a in Equation (4.8) was calculated by evaluating the integral using monte carlo integration at a number of random points for u, v and $\vec{\omega_o}$, and recording the maximum value encountered. This is done prior to rendering. In order to get a good estimate of the normalization factor a in Equation (4.8), a *low discrepancy* sequence of points was chosen instead of purely random points. A low discrepancy sequence is characterized by the fact that the points are more evenly distributed in space than a random sequence of numbers. Such a sequence is often used for numeric Monte Carlo integration since it offers a faster rate of convergence than the use of random numbers (Niederreiter, 2014). Here, however, the low discrepancy sequence is instead used to choose the points which are used for finding the maximum. The *Halton* low discrepancy sequence was chosen for the implementation, since it is easy to implement and offers good convergence as long as the number of dimensions d is relatively low. In this case, $d = 4$, since u and v has one dimension each and the direction $\vec{\omega_i}$ can be represented as a two dimensional point on the unit hemisphere. Figure 4.8 shows a comparison of 1000 points from the Halton sequence and from a random number generator. It is apparent that the points from the Halton sequence fills the plane in a more uniform manner than the random sequence of points does.

4.4.6 Integrating the Shader With Different Rendering Engines

The cloth shader was integrated into three different rendering engines, Mitsuba, V-Ray and the renderer developed in Chapter 5 below. The reason for this is that all three renderers have been designed with very different set of goals in mind.



(a) 1000 points from the Halton sequence.



(b) 1000 points created using the random number generator in Matlab.

Figure 4.8: A comparison of a sequence of points generated using a) the Halton sequence and b) random numbers. The points from the Halton sequence covers the plane in a much more uniform fashion than those from the random sequence.

Mitsuba is developed for academic purposes, with the goal of being easy to extend with new functionality. For this reason, it was used for the initial development of the shader. To interface with Mitsuba, Blender was used. The *mtsblend* add-on, (Styperek, n.d.) which allows Mitsuba to be used as a renderer inside Blender, was modified in order to support the parameters of the cloth shader.

The main goal of V-Ray, on the other hand, is to be easy to use for a 3D artist, but not necessarily for a developer. It is also the software used for all rendering at ICOM. For this reason, it was used for the final implementation of the shader. Developing plugins for 3ds Max can be a time consuming process, since the entire application needs to be restarted in order to reload the plugin. To circumvent this problem, the core part of the shader model was separated into a dynamically linked library which could be recompiled without restarting the entire application.

The goal of the renderer developed in Chapter 5 is to allow ray tracing to be performed at interactive rates. The integration of the cloth shader with this renderer is discussed in Section 5.4.7, after the relevant concepts of the renderer have been introduced.

4.4.7 Developing a General API for the Shader

Since the implementation of the cloth shader was meant to support multiple renderers, the core of the shader had to be generalized in order to be independent of any specific code base. This allowed the plug-ins for the renderers to only contain minor engine specific pieces of code, which in turn called the generalized shader code. The implementation was done in C, to ensure flexibility and allow the shader to be compiled in both C and C++ projects.

The intended use of the generalized API is as follows. During its initialization phase, the renderer will load the shader parameters from a user interface and store them in a variable of type `wcWeaveParameters`. This struct also includes some members that need to be set by calling `wcWeavePatternFromFile`, which reads the weaving pattern from a file, as described in Section 4.4.1 above. During rendering, when the BRDF needs to be evaluated, the function `wcGetPatternData` is used to get information about the pattern at the current shading point and the functions `wcEvalDiffuse` and `wcEvalSpecular` are used to calculate the amount of diffuse and specular reflection at this point, respectively. In order to call these functions, some data from the renderer is needed. The struct `wcIntersectionData` needs to be filled out with the u and v coordinates of the shading point, as well as the $\vec{\omega_i}$ and $\vec{\omega_o}$ directions. An example of how the API for the generalized shader is used can be seen in Listing A.1.

4.4.8 Adding Variation Through the Use of Noise

The reflection contains no source of imperfections and will result in perfectly identical highlights on each thread of the cloth. However, real fabrics are never perfect, and a bit of variation can add a lot of realism to the appearance of the material. This variation is introduced into two areas of the shader.

Firstly, noise is added to the highlights defined in Section 4.4.3. The surface is divided into small squares of uniform size. The size of each square is defined by the *fineness* parameter in the shader. The Tiny Encryption Algorithm (TEA) (Wheeler & Needham, 1994) is used as a hashing function to sample random values which are used to vary the highlight in each square. The use of a hashing function ensures that the same noise pattern occur at the same positions if the same scene is rendered again. The sampled value is transformed from the interval $[0, 1]$ into an exponentially distributed random value by taking its negative natural logarithm. For a uniformly distributed random variable X, the expected value will be

$$E[-log(X)] = \int_{-\infty}^{\infty} -log(x) f_X(x) dx = \int_0^1 -log(x) dx = 1.$$

In other words, the expected value will be 1 and by multiplying it with the highlight intensity means that the overall brightness of the material will remain unaffected. On average, the brightness will remain the same, and this noise variation should not affect any physical correctness of the underlying model.

Secondly, a low frequency variation along each yarn is introduced. Some fabrics have characteristic streaks along the warp or weft threads when viewed from afar. These streaks arise because of variations in color along the lengths of the individual yarns. To simulate this, noise can be used to vary the color along the length of the yarn. Using random numbers would be much too erratic to give the desired result. Instead, a smoother noise is required, such as *Perlin noise*, which is a common tool in the field of computer graphics. Many reference implementations of Perlin noise exist, such as the one by Ken Perlin himself (2002a). The noise can be given more detail by superimposing multiple samples of the noise at different scales. An example of this can be seen in Listing A.3, Appendix A.

4.5 Chapter Summary

This chapter has described the theory and implementation behind a shader which can simulate the appearance of woven cloth. The pattern in which the yarns are to be woven is read from a file and several parameters are exposed to allow the final appearance to be modified by the user.

First, the need and potential benefits of a shader for woven cloth were presented. Related work and current methods for simulating the appearance of woven cloth was reviewed, and several important papers used in this report were mentioned. Next, important foundational concepts where introduced, such as the basics of woven fabrics and how they are described in a weaving draft. The background chapter was concluded by a presentation of the concept of a Bidirectional Reflectance Distribution Function, BRDF.

The development of the cloth shader described in this chapter can be divided into a few key steps. These are the reading of a weaving pattern, the geometrical representation of the yarn, the model for reflected light intensity, the variation of the fabric through noise, and lastly the work of connecting the shader to different rendering engines.

The evaluation of the created shader is presented in the chapter on results, Section 6.2. Thoughts on the results and further improvements which can be made to the model are discussed in Section 7.2.

Chapter 5

Rendering– Interactive Ray Tracer

Rendering represents the final part of the content creation pipeline, where the physics of light is simulated to create realistic images. In this chapter, the aim is to develop a renderer that utilizes ray tracing, and is capable of producing images at interactive rates. In order to make this possible, the Graphics Processing Unit (GPU) is utilized for its large amount of processing cores.

This chapter will discuss the implementation choices in the design of the rendering tool. It begins by introducing technique choices and restrictions for this project. Secondly, sources used for understanding graphics is introduced, followed by development environment and file format choices.

Finally, the methods used to implement this project is presented, discussing the advantages of using these techniques.

5.1 Background

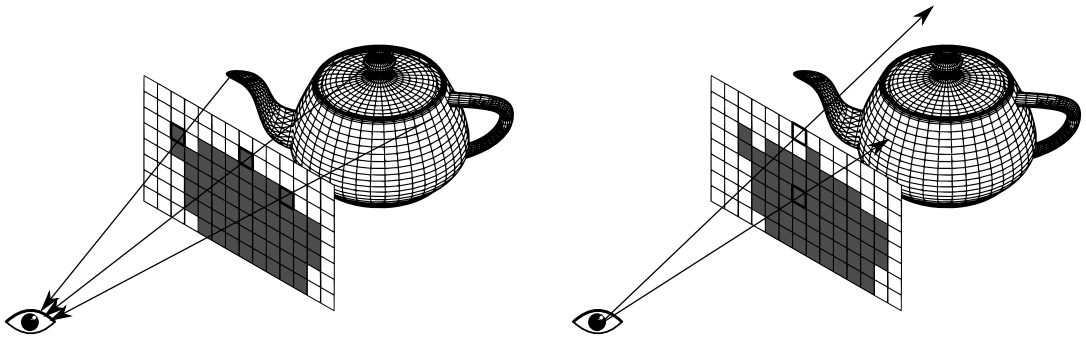
In computer graphics, a renderer has the task of generating the final image. This is done by keeping track of all objects and deciding which objects to display and how they should be coloured. This process is called rendering. As mentioned in Section 2.3, two common methods for rendering a 3D scene are *rasterization* and *ray tracing*, each having their own individual benefits.

In rasterization, each object in the scene is processed separately. The triangles of the mesh are projected onto the image plane, and each covered pixel is colored. In modern rasterization based renderers, the coloring of the pixels is determined by a *per pixel shader* or *fragment shader* (Akenine, Haines, & Hoffman, 2010, p. 30-31).

Ray tracing, on the other hand, processes each pixel of the image separately. The renderer creates *rays* which are fired into the scene. The renderer detects when a ray intersect with objects in the scene and calls upon a shader to evaluate the color of the current pixel, as can be seen in 5.1 on the following page. Furthermore, a ray tracing renderer often creates more rays from the intersection pointing in different directions depending on the material, thereby making the method recursive (Pharr & Humphreys, 2010). Since this method resembles the way light behaves in nature, ray tracing is capable of simulating complex light interactions, such as accurate reflections and indirect lightning (Rademacher, 2016).

Rasterization is a very fast method of rendering scenes, which makes it useful for interactive applications. However, it requires extra effort in order to achieve certain realistic light effects. Ray tracing, on the other hand, is usually a much slower technique, taking much longer to produce an image, but often gives more realistic scenes.

In practice, the areas in which the two techniques are applied in are split: games and other real-time applications almost exclusively use rasterization, while non real-time work,



(a) Rasterization *(b)* Ray tracing

Figure 5.1: In rasterization, for each object in the scene, the pixels being encompassed are filled and then colored using a fragment shader. In ray tracing, a ray is created going from the current view point through a grid representing the image plane. When an intersection between the ray and an object occurs, a shader corresponding to the assigned material of the object is called upon to evaluate the color of the pixel.

such as animated movies, or the product images produced by ICOM, often use ray tracing.

The *Graphics Processing Unit* (GPU) is a piece of hardware specifically designed to accelerate certain graphics related tasks. Specifically, its many cores makes it very good at performing parallel tasks. Interestingly enough, the GPU has traditionally been ill-suited for ray tracing. In fact, most ray tracing up until recently has been performed on the Central Processing Unit (CPU) (Pharr & Humphreys, 2010, p. 993). GPUs have generally been designed around rasterization, and important programming-concepts such as recursion have not been supported on GPUs.

Recent pushes by the graphics industry has resulted in graphics hardware capable of performing more general processing, while still utilizing the massive parallel processing power that a GPU provides. NVIDIA CUDA, for example, is a platform for running general purpose code on the GPU. Since CUDA's 2010 release, recursion has been implemented, allowing for algorithms such as ray tracing to be executed on the GPU. On top of this platform, NVIDIA also provide the OptiX ray tracing engine (Nvidia Software, 2015a); a set of tools and APIs that can be employed in order to utilize the GPU for ray tracing.

5.1.1 Using OptiX for GPU Ray Tracing

NVIDIA OptiX is a graphics engine built on top of the CUDA framework that is designed so that ray tracing can efficiently utilize the full power of the GPU. The OptiX framework provides an abstract model of a ray tracer, with an API that can be used in the development of ray tracing programs. The frameworks' ray tracing model is abstract in the sense that many of the key components are left to be implemented by the developer.

Some of the key components that can be implemented by the developer using OptiX include the production of rays, detecting intersections, and how these intersections should be handled. These are implemented using the CUDA programming language. At runtime, these programs will be executed on the GPU (Nvidia Software, 2015b).

Using these components, the framework can take care of many of the details involved in getting the code to the GPU, handling recursion and the communication between the different components. In addition, OptiX provides common data structures and algorithms that are used in ray tracing.

5.1.2 Executing Code on the GPU

The GPU have traditionally been designed towards rasterization. Recently, however, advances have been made in order to utilize them for other tasks. The NVIDIA CUDA Compiler Driver (NVCC) which is designed for running arbitrary code on the GPU is one of these advances.

The details of compiling CUDA code can be quite intricate, and NVCC has been designed to hide these intricacies from the users. It provides a frontend which allows compilation of projects in a way which is familiar to users of common C or C++ compilers (Nvidia Software, 2015c).

As can be seen in Figure 5.2, code is written in a .cu (CUDA) file and translated into a .ptx (parallel thread execution) file which contains machine code derived from the CUDA code. PTX defines a virtual machine and instruction set architecture for general purpose parallel thread execution. PTX programs are translated to the target hardware instruction set at install time. The PTX-to-GPU translator and driver enable a NVIDIA GPU to be used as a programmable parallel computer (Nvidia Software, 2015d).

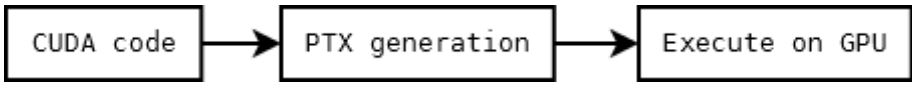


Figure 5.2: CUDA pipeline

The CUDA file defines all processes that will be executed on the graphics card; this is where the core of the ray tracing is implemented. It is crucial that this is the case, since ray tracing is the biggest bottleneck in rendering, and making the process parallel on the GPU as opposed to running on the CPU increases performance drastically.

5.2 Related work

Historically, ray tracing has been executed on the CPU. The article by Parker et al. (1999) discusses methods to implement ray tracing at an interactive rate as early as the late 90's by performing the ray calculations in parallel on several shared-memory supercomputers.

However, as presented by Wald and Slusallek (2001), "expensive, shared-memory supercomputers are not readily accessible". Instead, the authors present algorithms to speed up rendering.

More modern related works include that by the Theoretical and Computational Biophysics Group (2016), where molecules are rendered interactively using OptiX GPU ray tracing, and the work by Pellacini (2016), which is a renderer capable of both OpenGL rasterization and OptiX ray tracing. Also, while not related to OptiX, it is worth mentioning that V-Ray has an interactive ray tracing renderer implemented in 3ds Max that can run on either the CPU or the GPU (ChaosGroup, 2016).

5.3 Restrictions

The initial plan of this project was to implement an interface which could read a scene and render it using both OpenGL rasterization and OptiX ray tracing. The user would be able to press a button to switch between rasterization and ray tracing, demonstrating the differences between the two rendering techniques. However, due to time constraints, this was not feasible. Instead, it was decided to implement a rendering tool that only uses ray tracing.

Recall from Section 2.2 that an object's surface may have different materials applied to it. Because of the limited timeframe of this project, no way of adding different materials to

different parts of the same object was implemented. It is however possible to load multiple objects and apply a material of choice to each object.

Global illumination is a feature that takes into account light that is not directly emitted from a light source. Areas that are not hit by direct light will appear completely dark unless this is implemented. In the end, it was not implemented, since it is very computationally heavy due to the fact that it has to calculate light reflections even on non-specular surfaces or surfaces which may not be lit. With this in mind, it has been determined that it does not fit well into the idea of a fast rendering environment.

5.4 Method

The aim of the rendering part of the project was to create a prototype of a tool that can quickly visualize scenes used in production. This involved creating a GPU-accelerated renderer. The primary goal of the renderer was for it to produce as realistic images as possible, and to do this at an interactive rate. Utilizing the OptiX framework for ray tracing, a renderer was built in such a way that it was fast enough to work without major delays.

5.4.1 OpenGL Tutorials

The learning tool used to get the necessary knowledge to proceed with the project was the OpenGL tutorials from the Chalmers course *Computer Graphics - TDA361* (Assarsson, 2015). These tutorials were chosen as a learning tool because they provided an excellent introduction to the fundamentals of computer graphics, which was necessary to proceed with the project.

While all the tutorials were found useful, some of them were more important than others. One of the tutorials was about the camera and animations in 3D graphics, which covered how one can use spherical coordinates to rotate the camera in a 3D environment in an appealing way which was later used in the visualisation tool. Another useful tutorial introduced path tracing. It covered some useful concepts, such as shadow rays (explained in Section 5.4.6) and the rendering equation (Pharr & Humphreys, 2010, p. 908-911).

5.4.2 Visual Studio and CMake

The renderer was written in a combination of C++ and CUDA in order to maximize performance and allow for compatibility with the OptiX framework. The largest obstacle with this was that since a multitude of packages and support code was used in the project, the code itself was split between several files. In order to keep track of everything and link files together correctly, Microsoft Visual Studio 2013 (VS) was used. VS is also a requirement to be able to compile and run the OptiX sample code segments, which was a great starting foundation to build a new OptiX project upon. The OptiX samples provide several useful packages and code segments. Amongst these packages, the sutil project is found, which provides a multitude of methods that is utilized by the final product, including the Mouse and OptiXMesh files which help control the mouse and import objects, respectively.

In order to utilize C++ libraries, they must be linked together properly in VS; linking allows for methods to be used across different CPP files. For all of this to be connected correctly, the OptiX sample code is provided with several CMake files, designed so that the different libraries and projects are weaved together properly in the VS solution. For this project, a series of custom CMake files were designed, based on the OptiX sample files, so that only the renderer and the necessary support libraries were included.

5.4.3 Object File Format

While there are many file formats for 3D meshes, the Wavefront OBJ file format was used in this project. An OBJ file contains relevant information for describing one or more triangle meshes. It contains the coordinates of all the vertices, vertex normals, texture coordinates, and which vertices forms which triangles. A single OBJ file can also contain several meshes at once, which gives a designer the option to design an entire scene in any 3D modeling program and export the scene as a single OBJ file. It was decided to use the OBJ file format as input to the renderer, as these files contains all the needed information, are widely used and supported by most pieces of software. It was mentioned in Section 5.3 that one of the restrictions of the project is that only one material will be applied to each object, which means that a scene will have to contain multiple objects in order to use more than one material.

5.4.4 Camera Movement

Ideally, the camera would move using spherical 3D coordinates as introduced in the OpenGL tutorials. In terms of the spherical coordinates in Figure 5.3, mouse movement along the x-axis of the screen changed the θ-angle of the camera and mouse movement along the y-axis of the screen changed the camera's ϕ-angle.

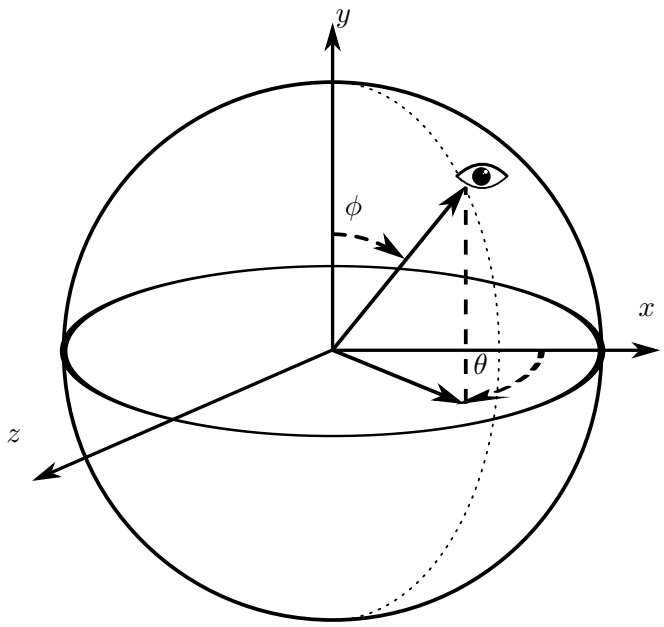


Figure 5.3: The spherical coordinate system used for camera rotation. The origin is defined as the lookat-point and the eye represents the camera.

The camera is defined by three variables: `eye`, `lookat` and `up`. `eye` is the position of the camera, `lookat` is the point it is looking at, and `up` is a vector which defines the upward direction relative to the scene. To make sure that none of the other camera functions were broken, for example zooming and moving around the camera, the new algorithm for camera rotation was designed to also use these three variables. To make the implementation of the camera rotation easier, `up` was locked so that it always pointed along the y-axis.

Let L be the `lookat` point and E be the `eye` point, defined as

$$L = \begin{bmatrix} x_L \\ y_L \\ z_L \\ 1 \end{bmatrix}, E = \begin{bmatrix} x_E \\ y_E \\ z_E \\ 1 \end{bmatrix}$$

The fourth dimension of these points have several uses in computer graphics, but the important part here is that it allows a translation matrix to be created. As the goal is to rotate the camera around the `lookat` and not the origin, the first thing to do is to translate the system so that the `lookat` is in the origin. This is done by multiplying the `eye` vector by the translation matrix

$$T = \begin{bmatrix} 1 & 0 & 0 & -x_L \\ 0 & 1 & 0 & -y_L \\ 0 & 0 & 1 & -z_L \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

After translating the coordinates, the rotation can be performed as if `lookat` was in the origin. For the θ rotation, the `eye` is multiplied with the following rotation matrix

$$R_\theta = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

where θ is how far the camera should rotate. This is a standard rotation matrix (Råde & Westergren, 2004, p. 111) extended to four dimensions. For the ϕ rotation, first a θ rotation is done to make `eye`'s z-coordinate be 0. The vector $\vec{v} = E - L$ determines the angle of this rotation. The θ-angle to rotate by will be $-\arctan\left(\frac{z_v}{x_v}\right)$. As $\tan(x)$ is π-periodic, π needs to be added to this angle when θ is greater than $\frac{\pi}{2}$ (as the range of arctan is $[-\frac{\pi}{2}, \frac{\pi}{2}]$), which is when x_v is negative. Putting this into the rotation matrix results in the matrix

$$R = \begin{bmatrix} \frac{1}{\sqrt{(\frac{z_v}{x_v})^2+1}} & 0 & \frac{z_v}{x_v\sqrt{(\frac{z_v}{x_v})^2+1}} & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{z_v}{x_v\sqrt{(\frac{z_v}{x_v})^2+1}} & 0 & \frac{1}{\sqrt{(\frac{z_v}{x_v})^2+1}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

with the elements $(1,1)$, $(1,3)$, $(3,1)$ and $(3,3)$ multiplied by -1 when x_v is negative. After this rotation, the ϕ rotation can be performed by multiplying with a matrix which rotates around the z-axis. As the θ rotation performed earlier was a rotation around the y-axis, both rotation matrices are similar. The rotation around the z-axis is done by the following matrix:

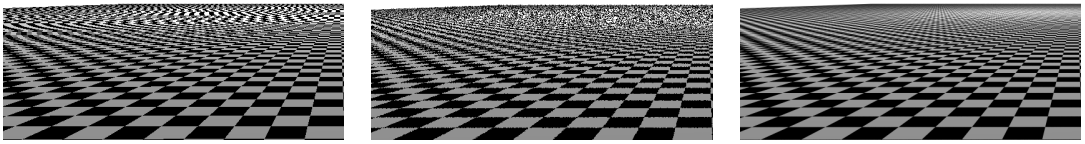
$$R_\phi = \begin{bmatrix} \cos\phi & \sin\phi & 0 & 0 \\ -\sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

After performing this rotation, `eye` is multiplied with the inverse of R to get back to the correct θ angle. After the rotations are done, the system is translated back to the origin by multiplying with the inverse of T.

5.4.5 Anti Aliasing

In a general ray tracing algorithm, a ray is fired and mapped to its one respective pixel, coloring the pixel. Unless handled properly, this will display an image with very hard edges and, when moving a detailed shaded surface, flickering can occur (see Figure 5.4a). This phenomenon is called *aliasing*, and is, according to Jimenez, Echevarria, Sousa, and Gutierrez (2012), "one of the longest running problems in computer graphics".

Anti Aliasing aims to prevent this phenomenon and can be implemented in a number of ways. *Supersampling* is one such solution, where multiple rays are cast per pixel and the average color contribution from each ray is calculated. The problem with this technique is that it does not scale well, which is a important factor for a fast, interactive renderer. Taking 16 samples per pixel will end up with a much clearer image, but it will also take 16 times as long in order to compute, as the time it takes to compute a frame scale linearly with the amount of rays which are fired.



(a) Visible aliasing.

(b) Antialiasing using subpixel jittering.

(c) Antialiasing using supersampling

Figure 5.4: A comparison of the same image with aliasing, subpixel jittering and supersampling.

A much faster solution to this problem is *subpixel jittering*. For each pixel's ray, a small random offset in the xy-plane is applied, causing a noise effect, as shown in Figure 5.4b.

Since the renderer is fast enough to handle a certain grade of supersampling while maintaining an interactive framerate, both of the aforementioned methods have been implemented. There are many ways of implementing supersampling, and using subpixel jittering to sample several locations within a pixel is one of these. The result of using supersampling with subpixel jittering can be seen in Figure 5.4c.

5.4.6 Calculating Shadows

Shadows are an important part of rendering a realistic image. The basic concept behind rendering shadows in a ray tracer is quite intuitive. A ray is sent towards the light source, and if this ray intersects any object before it reaches the light source, the point is in shadow. Otherwise, it is fully lit.

The simplest form of light source is *point lights*, which are defined by a single point in space. While point lights are easy to work with, they always produce shadows which are perfectly sharp, which tends to give the resulting images an artificial look. The reason that the shadows are sharp is that all the light originates from a single point in space. This is the same as in reality, where a small light source cast shadows which are sharper than a large light source. However, a point light is the result of shrinking a light source down to an infinitesimally small point, and this produces a perfectly sharp shadow (Suffern, 2007).

To alleviate this problem, all light sources in the ray tracer were treated as spheres, instead of just points, with regards to calculating shadows. Whenever a shadow ray was to be calculated, a random location on the surface of each light-sphere was sampled, and the shadow ray was sent towards these locations. As can be seen in Figure 5.5, this leads to a soft shadow. This approach, however, relies on the use of multiple shadow rays per pixel. Using a single shadow ray per pixel would result in very noisy shadows.

5.4.7 Utilizing the Cloth Shader

A special material was defined in order to utilize the shader for woven cloth introduced in Chapter 4. With the help of the cloth shader, the renderer gains the ability to emulate woven fabrics in a realistic way. Since the shader was written in C, it could be compiled into CUDA using NVCC without problems.

(a) Shadows from a point light *(b)* Shadows from a spherical light

Figure 5.5: A comparison between the shadows cast by a point light *(a)*, and a spherical light *(b)*. The shadows cast by the point light are perfectly sharp since all the light originates from the same point in space. Meanwhile, the spherical light results in soft shadows, since the obstacle might block the light source partially.

When a ray intersects an object which has been assigned the cloth shader, the incident and outgoing directions were transformed to the coordinate system used by the shader, and the relevant functions were called to calculate the specular and diffuse reflection, as specified in Section 4.4.7.

While the shading could be done on the GPU, the functions that read a pattern file had to be called from C++ instead of CUDA. This posed a problem, since the pattern data is returned in an array which is allocated on the heap. The GPU has its own memory and cannot access data which is allocated by the CPU. Normally, a pointer would keep track of the allocated memory, but this approach did not work across the boundary between CPU and GPU. Instead, the pattern matrix had to be sent to the shader separately from all other parameters.

Listing A.4 contains the core part of the OptiX Material which calls the cloth shader.

5.5 Chapter Summary

This chapter has presented the general structure of CUDA compilation used in the OptiX framework and the structure of a ray tracing renderer. The created renderer can visualize any combination of objects and materials, which are defined in code and can be modified to represent any scene with restriction on the types of materials which can be applied.

At the beginning of the chapter, the reasoning behind choosing ray tracing for an interactive renderer was discussed. Next it is explained why ray tracing has historically been slow, and how NVIDIA has worked towards making this process parallel on a GPU resulting in the OptiX framework. Following this, a brief presentation of related works by others is put forth. Restrictions for the project is also introduced.

Subsequently, a brief description of the OpenGL tutorials is discussed, followed by a brief presentation of the IDE and file format choices. Finally, a walkthrough of the methods that are implemented in order to achieve the goals of the project are discussed, such as camera movement and visual improvements to the renderer, and a brief discussion about integrating the cloth shader project into the OptiX renderer.

The results of this project can be found under Section 6.3, where the appearance of the rendered results and the performance of the renderer is analyzed and measured. These results are discussed in Section 7.3.

Chapter 6

Results

In this work, three different tools have been designed. A tool for 3D-modeling of rattan furniture, a shader for woven cloth, and an interactive rendering engine that utilizes the GPU to perform ray tracing. In this chapter, the results from each of these projects are presented.

6.1 Tool for the Modeling of Rattan Furniture

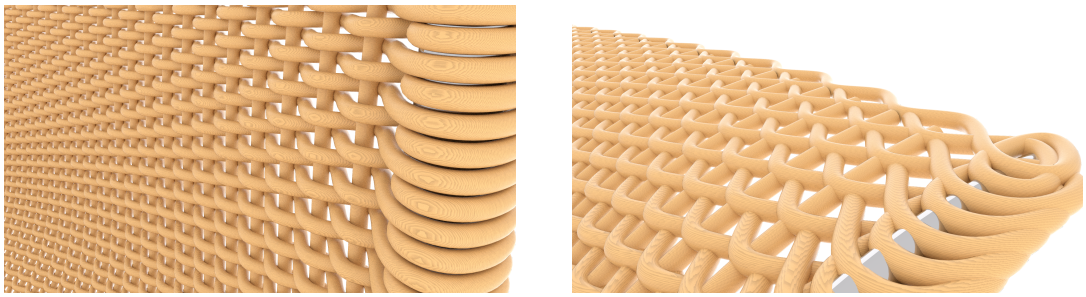
In this section the output from the final tool is presented. Both detailed close-ups and a finished product design are shown. The tool is added as a modeling function in Blender. It has a setting for the thickness of the strands and a button which consums the current selection as input and runs the generation.

6.1.1 Visual Appearance

Figure 6.1 contains an example of the output from the rattan tool. In this figure, the tool is used to visualize a simple chair design which can be compared with the image of a real rattan chair in Figure 3.1. Although the chairs are very different, the structure from the interwoven rattan strands is similar.

Figure 6.1: A chair design created with the rattan tool.

Figure 6.2 shows close-ups of two parts of the chair. Figure 6.2a was chosen to show a part of the model where the tool creates output as intended. Figure 6.2b displays a part of the model where the tool fails to create the intended output.



(a) Example of good results. *(b)* Example of poor results.

Figure 6.2: Images showing examples where the rattan tool works well and not so well.

The model was created by a user with little 3D modeling experience. The first step was to create the rattan surfaces for the seat and backrest of the chair using the tool. Secondly, the frame was created as a single NURBS curve which was matched to the frame connections generated by the tool. Lastly, the small rubber feet were modeled with cylinders.

6.2 Shader for Woven Cloth

Evaluating the quality of the cloth shader involves looking at both the performance and the visual quality of the results. In this section, the memory and CPU usage of the shader are investigated, as well as its ability to recreate the look of different types of fabrics. This evaluation is done by considering two different physical cloth samples and comparing the results of the shader both with photos of the samples, and with the V-Ray shaders currently in use at ICOM. Furthermore, a short study is made on the range of materials that can be reproduced using the shader.

6.2.1 Appearance at High Magnification

Even though the shader is not meant to be used at close ranges, it is nonetheless useful to inspect it at high magnification in order to see its behaviour. Figure 6.3 on the next page shows an illustration of a yarn segment represented by the shader. In the image, the individual filament threads are visible as they twist around the yarn. In the shader, however, the number of filaments is assumed to go to infinity, which would give the cylinder a smooth surface. Figure 6.4 on the next page shows a comparison between the specular highlight from the yarn segment in Figure 6.3 on the next page, and a yarn segment in the cloth shader. While the result from the shader is less detailed, it nonetheless has a similar highlight shape as the reference geometry.

Figure 6.5 on the next page shows a scene with the cloth shader at different magnification levels. At the highest magnification, the cylinder representation and highlights on each individual yarn segment are visible. These highlights have a large impact on the final appearance of the cloth.

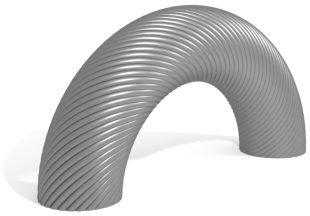
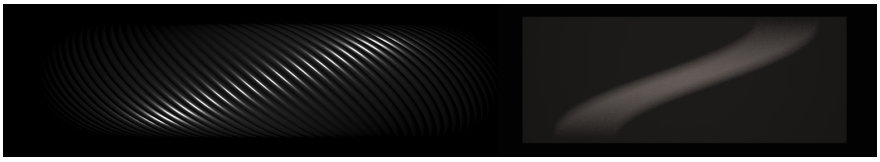


Figure 6.3: The geometry of a yarn segment, which is made up of twisted filament threads.



(a) Rendering of a yarn segment using explicit geometry.

(b) Rendering of a yarn segment using the cloth shader

Figure 6.4: A comparison between *(a)* a rendered image using a geometrical representation of a yarn segment and *(b)* the cloth shader.

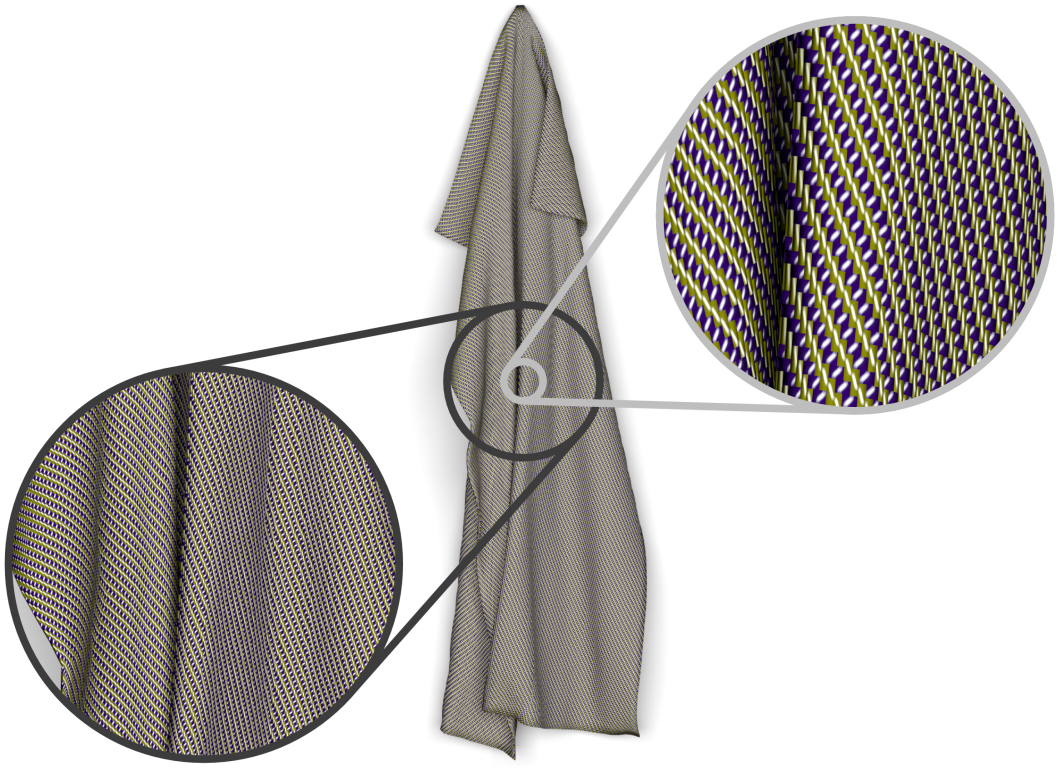
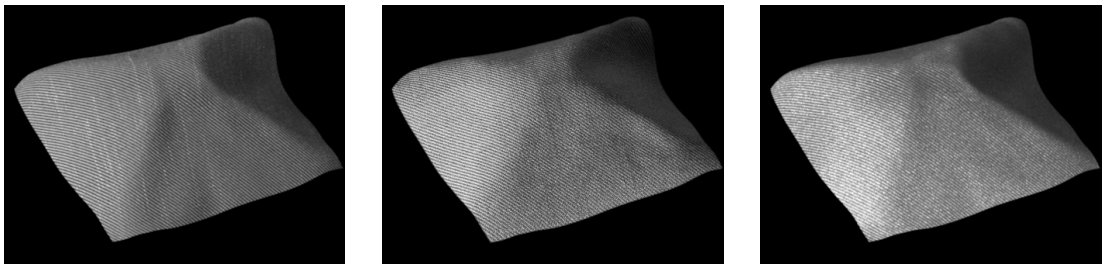


Figure 6.5: A scene with the cloth shader applied to the fabric mesh. Different degrees of magnification are included to illustrate how the shader works. At the highest magnification, the warps (in blue) and wefts (in yellow) are clearly visible, as well as a highlight on each yarn segment.

6.2.2 Comparison With Physical Cloth Samples

Two physical sample fabrics, “Nordvalla” and “Skiftebo”, and their corresponding V-Ray shaders were provided by ICOM in order to perform tests. To evaluate the visual results of the shader, its appearance was compared to photographs of the reference cloth samples under controlled conditions, as well as to the V-Ray shaders.

In order for the cloth to have a predictable shape in the photographs, a template that the cloth could be draped around was 3D printed using a consumer 3D printer. Camera and light parameters were noted and the scene was recreated in a 3D-package. The physical cloth samples did not fit perfectly on the template and some adjustments had to be made to the 3D model in order for the rendered images to match the photographs. Using this scene, images were rendered both with the ICOM shaders and with the cloth shader. A comparison of the ICOM shaders, the physical sample, and the cloth shader developed in this project are shown in Figures 6.6 and 6.7. Note that some of the visual differences between the photos and the rendered images are the result of difficulty in accurately recreating the geometry and lighting in the photos.



(a) ICOM shader. *(b)* Reference photo. *(c)* Cloth shader.

Figure 6.6: A comparison of the “Nordvalla” fabric with *(a)* the shader currently used at ICOM, *(b)* a reference photo and *(c)* the cloth shader presented in this report. Notice that the highlight at the left side of the cloth, which our shader simulates, is completely missing from the ICOM shader.

(a) ICOM shader. *(b)* Reference photo. *(c)* Cloth shader.

Figure 6.7: A comparison of the “Skiftebo” fabric with *(a)* the shader currently used at ICOM, *(b)* a reference photo and *(c)* the cloth shader presented in this report. It is clear that the ICOM-shader fails to correctly capture the reflections that appear in the reference photo and in the cloth shader.

The fabric in Figure 6.6, ”Nordvalla”, is a twill weave fabric. The photo shows a strong specular highlight on the left side of the fabric. This reflection is completely missing from Figure 6.6a, the shader used at ICOM. However, the cloth shader, shown in Figure 6.6c, accurately simulates this phenomenon.

In Figure 6.7, the fabric ”Skiftebo” is shown. It consists of a plain weave pattern which shows an interesting structure by having random variations in the yarn sizes. An attempt at recreating these irregularities has been done in both the shader used at ICOM (Figure 6.7a) and the cloth shader (Figure 6.7c). The variations in the cloth shader were

accomplished by utilizing the noise functions introduced in Section 4.4.8 and by manually altering the weaving pattern.

Range of Reproducible Materials

In order to test the expressiveness of the cloth shader, a few materials with different visual characteristics were recreated. Figure 6.8 show example of a satin type fabric and a tartan cloth created with the cloth shader. These images are the result of different combinations of parameters and weaving patterns. While there are no physical samples to compare these results to, they still serve to illustrate the range of materials the shader is able to reproduce to a reasonable degree of realism.

(a) A rendering of satin. *(b)* A rendering of tartan cloth.

Figure 6.8: Some examples of the range of materials which the cloth shader is able to reproduce.

6.2.3 Performance of the Cloth Shader

A performance benchmark was created to test the computation time of the cloth shader. For each test, the benchmark was run 200 times, and the average value and standard deviation of the computation time were recorded. Each run of the benchmark consisted of sampling the shader 100 000 times with random values of $u, v, \vec{\omega_i}$ and $\vec{\omega_o}$. The accumulated computation time of the calls to the shader were recorded using the Windows high resolution performance counter.

Measurements were made of the computation time as a function the size of the weaving pattern. In Figure 6.9, the relative benchmark times for random patterns of different sizes are shown. The variation between the different sizes is quite low, and there is no general upwards trend in the data. A conclusion can be made that the computation time does not scale noticeably with pattern size.

This result is backed by the fact that the only place in the code where the pattern size comes into play is when the size of a segment rectangle is to be computed, see section 4.4.2, and this function scales with the length of the segment, not with the size of the pattern.

Figure 6.10 shows a comparison between the computation time of the cloth shader and the popular *GGX* BRDF introduced by Walter, Marschner, Li, and Torrance (2007). This is a well used BRDF model for specular reflection which can serve as a baseline for the performance of a general purpose shader. The reason that GGX is used for the

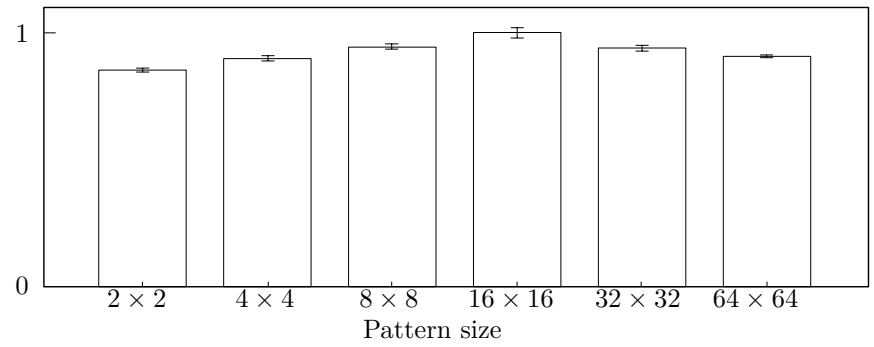


Figure 6.9: Relative computation time for different pattern sizes. Apart form the peak at 16×16, there is no general upward trend, and the computation time can be assumed to be decoupled from the size of the pattern.

performance comparison, instead of the V-Ray shader used by ICOM, is that an existing implementation of GGX could easily be integrated into the performance benchmark, while the ICOM shader would require the entire V-Ray runtime.

While GGX is often coupled with some sort of diffuse reflection in order to be a complete shader, the performance impact of calculating the diffuse reflection is negligible. The

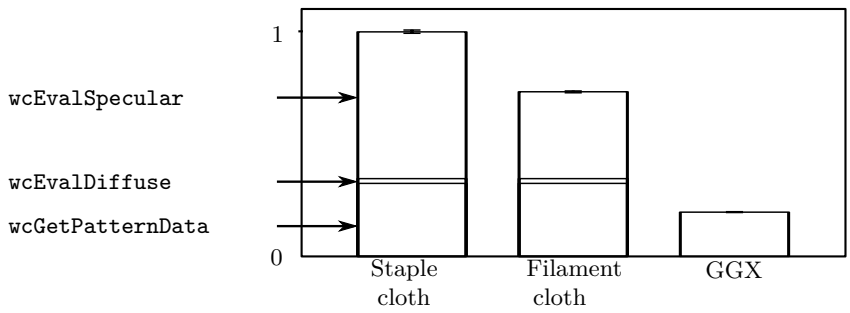


Figure 6.10: Relative computation time for the cloth shader compared to the GGX BRDF. The lower part of the cloth shader bars represent the function `wcGetPatternData`, the middle part represents `wcEvalDiffuse` and the upper bar is the time spent in `wcEvalSpecular`.

computation time for the cloth shader is about four or five times that of the GGX BRDF. Please note, however, that the GGX BRDF is far less complex than the cloth shader, and that it is a well optimized algorithm.

The bars representing the cloth shader in Figure 6.10 are divided into three parts. The lower part represents the time spent in the function `wcGetPatternData`, the middle part is the time spent in `wcEvalDiffuse` and the top part represents `wcEvalSpecular`. Evidently, more than half the computation time is spent calculating the specular reflection, and the other big contributor to the computation time is `wcGetPatternData`. Furthermore, the figure shows that specular reflection from staple cloth is slower to compute than that from filament cloth.

6.3 OptiX Renderer

This section will present example images generated using the OptiX renderer. The images demonstrate its abilities to simulate the appearance of certain materials as well as handling

the geometry of more complex objects. Results from performance tests are also presented.

6.3.1 Rendering Different Materials

Six different materials have been defined for the purpose of this project: cloth, black, floor, wood, metal, and glass, where some materials are more distinct than others. Black material, for example, is the most basic material that can be made. It does not have any other properties apart from setting a solid color, representing a theoretical best-case as a form of control material. Metal and wood material have different grades of reflection, and the glass material has refraction effects. The cloth material maps a cloth pattern to a surface, communicating with the cloth shader defined in Chapter 4.

In order to do a comparison of how the different materials affect the performance of the renderer, a scene with a single object was rendered using the same camera perspective, settings and hardware but with different materials applied to the object. The resulting images can be seen in Figure 6.11 and the different framerates when rendering the images can be seen in Table 6.1.[1]

(a) Wood material *(b)* Metal material

(c) Glass material *(d)* Cloth material

Figure 6.11: A demonstration of four materials, applied to a teapot model.

6.3.2 Handling Complex Structures

Figure 6.12 contains a rendering of a rattan chair created by the rattan tool in Chapter 3. The framerate achieved here was 4.5 frames per second with 1 sample per pixel, where the rattan weave consists of over 500 000 vertices. Substituting the wood material for the black material, the framerate increases to over 20 frames per second, since the wooden material is slightly reflective. Note that a blue cow horse is visible in the background through the rattan material, displaying that light can pass through the weave in the OptiX renderer.

[1]The graphics card which was used for these FPS results was an ASUS GeForce GTX 660Ti. Running on a NVIDIA GeForce GTX 840M, the amount of frames rendered per second was roughly halved.

Material performance	
Material	FPS
Black	94.30
Floor	37.89
Cloth	23.56
Wood	20.17
Metal	16.79
Glass	4.61

Table 6.1: Framerate of scene with different materials applied to a teapot model. The parameters used for this test was 4 samples per pixel with a maximum amount of bounces of 128. For the shadows, a total of 8 samples per pixel were taken.

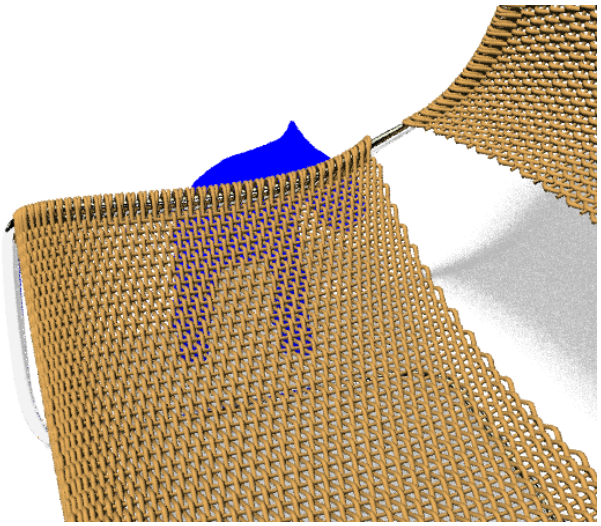


Figure 6.12: A rendering of the rattan chair used in Figure 6.1. A blue cow horse is visible through the rattan geometry (Creative Tools AB, 2011).

6.3.3 Shadows

As mentioned in Section 5.4.6, a soft shadow algorithm has been implemented in the renderer. Figure 6.13 shows a black teapot casting a shadow on a white floor, displaying the difference in quality when using different amounts of shadow rays; increasing the amount of shadow rays can fine-tune the quality at the expense of rendering speed.

6.3.4 Cloth Material

While the lighting and perspective is slightly off in Figure 6.14, it can be observed that the material itself looks very similar to the reference photo. The cloth shader displays a reasonable degree of realism, even when integrated with the OptiX renderer.

6.3.5 Anti Aliasing

In Figure 6.15, the difference in quality when using anti aliasing can be seen. Note the hard pixelated edges and the aliasing on the cloth surface. The second picture shows less of this behaviour after the anti aliasing techniques have been applied.

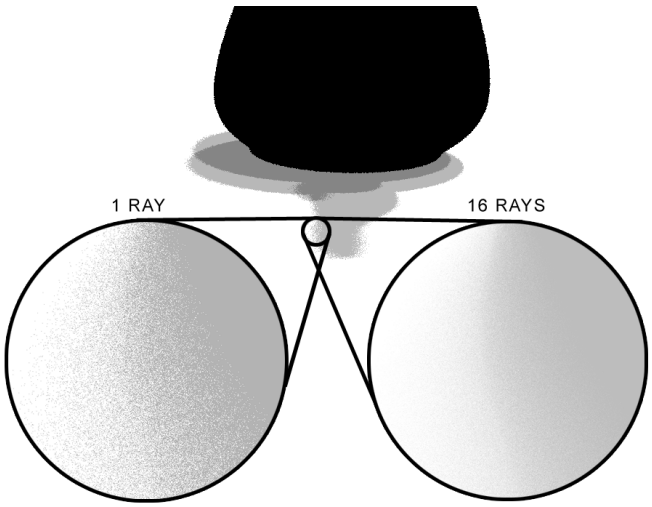
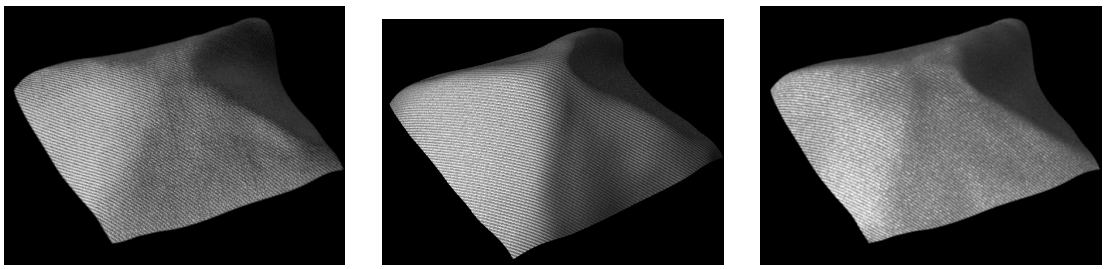


Figure 6.13: A comparison of soft shadows using 1 shadow ray per pixel and 16 shadow rays per pixel.

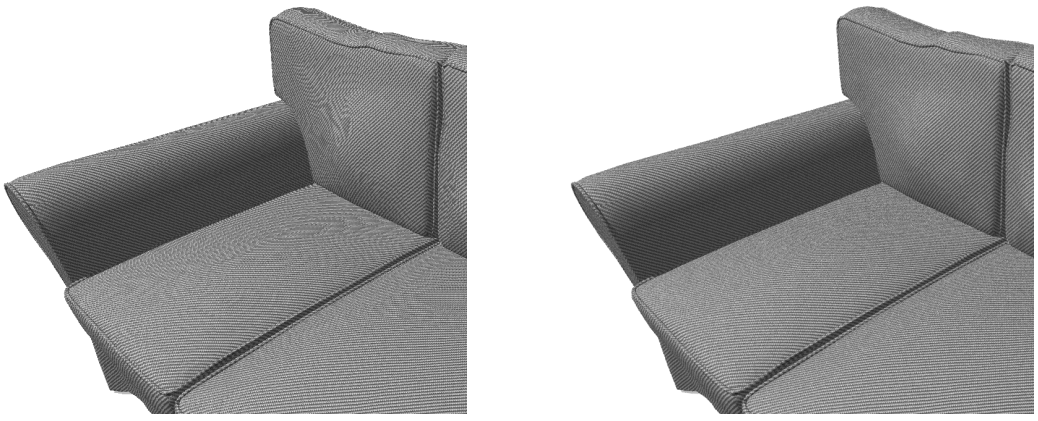


(a) Reference photo

(b) OptiX render of the cloth shader

(c) VRay render of the cloth shader

Figure 6.14: A comparison of using different renderers. Figure (c) uses the V-Ray renderer used to compare shaders in Figure 6.6 and 6.6, and acts as a comparison to the OptiX renderer. Note that Figure (b) is shot from a slightly different perspective and has different lighting.



(a) Without antialiasing

(b) With antialiasing

Figure 6.15: A comparison of the before and after result when applying a subpixel jitter to the rendering.

Chapter 7

Discussion

The use of the 3D image production pipeline to create photorealistic images can result in great benefits and added flexibility compared to traditional photography. The use of computer graphics does, however, still have plenty of challenges which prevent it from being used in certain contexts. One problem is the 3D modeling of complicated objects such as rattan furniture, another is the simulation of the realistic appearance of woven cloth and a third is the slow nature of the rendering process.

From the results in the previous chapter, it can be seen that it is possible to automate the process of creating detailed 3D models of rattan furniture. Secondly, it has been shown that existing models for the appearance of woven cloth (Irawan & Marschner, 2012) can be implemented and used to give good results in a production environment. Lastly, the results of the renderer show that the NVIDIA OptiX framework can be used to quickly preview materials in a ray-tracing environment, giving artists a faster feedback loop during work.

7.1 Tool for the Modeling of Rattan Furniture

As stated in Section 3.1 there is little previous work on procedural generation of woven geometries in similar scale to rattan or cane. This project set out to create a prototype of a tool which does this. The resulting tool can generate good results for a limited number of surfaces, see Section 6.1.

7.1.1 Visual Appearance

As Figure 6.1 demonstrates, the output from the rattan tool can be believable in an appropriate design. The output is usable in the background and in smaller scales. However it is not acceptable for professional productions. The details in Figure 6.2 show that the output is not believable in a bigger magnification, it is visibly artificial and regular. It appears very artificial but this could be alleviated with the introduction of noise in the position of control vertices. In Figure 6.2b the strands intersect each other which is not believable.

This model, however, was created by a user with minimal 3D modeling experience. In this light the tool achieved its purpose, to ease the creation of rattan-like geometry. There is potential for this tool to eventually be a useful part of the toolbox for 3D artists working with furniture modeling.

7.1.2 Usability for the Artist

The input to the tool are two quadric Bezier curves. They define the shape of the surface in 3D. This work similar to a *skinning* operation in popular 3D packages, one of the curves

is traced along the other to create the surface. This operation is well know to 3D artist so the tools is easy to pick up. However, since it is only one quadratic Bezier segment in each axis this limits the possible shapes of the output. The tool would be more usable with more degrees of freedom in the input.

The performance of the tool is satisfactory but not great. A faster tool would be better but the current calculation time does not impact negatively on the iteration time for the artist. The code is not optimized and it is possible that it could be made a lot faster.

7.1.3 Further Work

There are many possibilities for further work on this problem. Here, we introduce a few possible improvements to the tool that can be implemented in the future.

Organic Imperfections

The tool currently does not model any organic imperfections in the geometry. This is a key part in the appearance of real rattan furniture. As a start, this could be modeled by including noise in the transformation of each control point. Using Perlin noise, as described in Section 4.4.8, would probably improve the believability of the output.

Strand Intersections

In certain conditions the strands intersect with each other. This is not realistic and solving this issue would improve the output considerably. One approach would be to model the bending and intersections of strands using a physical model. Implementing ideas from Kaldor et al. (2008) could be a good starting point.

Usability

The usability of the tool can be improved. Right now the tool is simple to use but does not offer the level of control that is needed for professional content. It is necessary to give the artist more control but at the same time try to keep the interface as simple as possible. This would preferably be done by working together with professional 3D artists and iterating on the interface. A part of this would be to investigate if a better implementation could run as a real-time modifier. This would create a much shorter feedback cycle and therefore improve the usability of the tool.

7.2 Cloth Shader

From the results shown in Section 6.2, it is apparent that for some types of woven fabric, the cloth shader can produce very realistic results. The shader seems to be best suited for reproducing fabrics with a small and regular pattern. An example of this is the “Nordvalla” fabric, shown in Figure 6.6. Here, the rendered result using the cloth shader is very similar to the reference photo, showing features that appear to be completely absent in the shader currently employed by ICOM.

In Figure 6.7 the results of trying to mimic the appearance of the “Skiftebo” fabric are shown. This fabric features more large scale variation than “Nordvalla”, both in terms of color and structure. The rendered result sucessfully captures reflections and other similarities to the reference photo, but is still lacking in some areas. One limitation of the shader is that the color is the only parameter which is allowed to vary between individual yarns. Skiftebo suffers from the fact that the yarns in the model are defined on an evenly spaced rigid grid. In the result, some structural variation was added by manually altering the pattern and adding multiple wefts beside each other to simulate a

thicker yarn. However, as can be seen in Figure 6.7c this varaition repeats too regularly. Ideally, the user should be able to specify different parameters for the warp and weft threads and apply variation to all parameters such as size and reflective properties, rather than just color.

Additionally, some attempts were made to recreate fabrics with different visual appearances. Figure 6.8a and Figure 6.8b show examples of using the shader to simulate the appearance of satin cloth and tartan cloth, respectively. This is by no means a rigorous evaluation of the realism of the result, but it illustrates the range of fabrics that the shader is capable of producing while still being visually plausible.

7.2.1 Performance Comparison With General Purpose Shaders

From the performance measurements in Section 6.2.3, it was concluded that the cloth shader is about 4 times as demanding as the commonly used GGX shader. One reason for the difference in performance was first thought to be the fact that the cloth shader has to access memory in order to read the weaving pattern, but further investigation suggests that during the benchmark, the used memory might be small enough to fit in the CPU cache. This means that the memory access does not affect the computation time. Instead, the explanation probably lies in the fact that the calculation of the cloth shader involves more instructions than the GGX shader, among them several square roots and trigonometric functions, and little effort has so far been made to reduce the amount of computation it requires.

7.2.2 Conclusions Regarding the Results

The results from the shader show impressive results for the right types of fabric and can be used in conjunction with tools commonly found in a production environment. The shader does require more computation time than a general purpose shader would, but at the same time, it alleviates other hurdles, such as storing and managing a large library of high-resolution textures by instead having the shader generate the appearance of patterns from a simple text description. This also has the benefit of freeing up memory during rendering that would otherwise be used to store high-resolution textures.

Another interesting use for the shader could be during development of new fabrics and weaving patterns. A preview of a pattern being designed could be rendered without having to create a prototype weave or having to wait for a sample to be sent from a factory.

7.2.3 Further Work

The shader developed in this project serves its purpose as a basic implementation, but there are many improvements that could be done in order to make it a more viable tool for 3D image production. This section will present the most urgent changes in order to make the shader more expressive and easier to use.

Making the Cloth Shader Parameters Easier to Understand

So far, very little work has gone into trying to make the shader easier to control. However, three main things which could improve its usability have been identified. Firstly, many of the parameters in the current shader could be removed or combined. For instance, the noise is responsible for almost a third of the parameters and could be extracted out of the shader. The α and β parameters of the phase function could be combined into one single parameter since their sum is later normalized. Secondly, the parameters could be given names which are more descriptive for a non-technical user. For instance, u_{max} could be renamed to “bend amount”. Renaming the parameters in this way would be a

very simple change and would also make them easier to understand. Thirdly, the range of feasible parameter values are currently very hard to predict. Some settings, such as Δx, has values in the interval $[0, 1]$, while others, such as u_{max}, represents an angle with values in the range $[0, \pi/2]$. As mentioned by Burley and Walt Disney Animation Studios (2012), it is beneficial to instead let the usable range of all parameters range from zero to one.

Added Flexibility

In many rendering engines, such as V-Ray, is it common to allow the user to control many of the shader parameters using textures. A similar approach could be taken with the cloth shader, allowing the user to vary any setting, such as u_{max} or ψ, using a texture. Another important improvement to the model would be to allow different sets of parameter values for different yarns. Apart from the color, the warp and weft threads currently all have the same properties. This limits the amount of materials that can be recreated, since many fabrics have multiple yarns with different properties.

Finding a More Accurate Phase Function

The von Mises phase function used in the model for light scattering inside the yarn is a general model for forward scattering and while it gives results that look good enough, it has little physical grounding. It would be interesting to perform measurements on different types of yarn fibers to determine the shape of their phase functions. These types of measurements have been made for light scattering inside hair (S. R. Marschner, Jensen, Cammarano, Worley, & Hanrahan, 2003), which by now is a fairly well understood problem. However, the microscopic geometry of a yarn filament can vary significantly from that of hair, which means that the measurements made for hair is not directly applicable to cloth rendering.

Even more complexity is introduced when considering dyeing of yarn, which affects its scattering properties, or the fact that it is common to print patterns on the cloth, which adds a layer of paint to the filament and makes the scattering even more complex.

7.3 Optix Renderer

As seen in Table 6.1, the renderer achieves very acceptable rendering rate with some materials rendering in real-time. It is also apparent from this table that utilizing the cloth shader to apply a material is surprisingly fast, rendering at a higher rate than the wood material. This is also a good demonstration of how much reflectivity affects performance. Not surprisingly, the glass material fared worst, proving that refraction requires longer calculations.

7.3.1 Cloth Shader Integration

As can be seen in Figure 6.14, the cloth shader integration with the OptiX implementation fares well compared to reference images and other renderers. While it may look less realistic than the comparison images, it is worth noting that this result is still very impressive considering many features were left out in order for the image to render faster. At the distance displayed in Figure 6.14b, the renderer performs at a rate of over 15 FPS, with 4 samples per frame.

7.3.2 Rendering Rattan Furniture

As expected, objects with a large amount of vertices slows down the renderer. As can be seen in the rendering in Figure 6.12, the chair renders at a rate of 4.5 FPS, which is roughly as slow as rendering a glass teapot. Naturally, the rattan chair used here could be replaced with a lower resolution model, thus speeding up the process.

7.3.3 Further Work

Beyond the scope of what the project has achieved, there are many improvements to be made and features that can be added. This section will discuss some of these, including some features that were considered but was not completed due to time constraints.

Progressive Rendering

During rendering, the renderer currently first samples the scene a set amount of times, and then displays a static image. If the camera is completely still, the same image is rendered each frame and redundant ray tracing calculations are made, slowing down the computer and even user input. This could be solved in one of two ways: stop ray tracing calculations after a frame has been calculated (thereby saving both power and GPU usage) or remove the sample limit and increase the picture quality progressively.

By continuously rendering the scene and updating the image progressively, the image will become better and better after each iteration of samples. Moving the camera around will also become much smoother in the case that a user wants more than 1 sample per pixel, as camera movement will not have to wait for 2 or more full renderings to compute before changing perspective.

Usability

In its current state, the renderer is far from usable. Rendering settings, object definitions and object materials are all hard-coded. In order to change the settings, one would have to find the correct place in the code to input this data and recompile the renderer. The renderer could be improved by a graphical user interface that allows a designer to edit the settings in a more user friendly way, perhaps even changing the settings live while the renderer is running.

Defining Materials

Extending upon usability, it is difficult to implement custom material definitions. A designer need to know how to define shader algorithms in the CUDA files. A general all-purpose shader could be implemented that allows for a designer to apply rendering settings such as colors, textures, specific reflectiveness algorithms or refraction.

Beyond this, the renderer currently only serves its purpose as a proof of concept for different materials. In order to use the renderer as a working rendering environment, it would be useful for a designer to be able to apply several materials to different parts of a single object.

Anti Aliasing using the Halton Sequence

As presented in Section 5.4.5, the renderer uses a subpixel jitter in order to remove the aliasing effect. The subpixel jitter is random, which implies that it may or may not always guarantee the same quality in the final rendered image. Instead of a random offset, the renderer could use the Halton sequence, as explained in Section 4.4.5. However, care will have to be taken to implement this properly, as Hess and Polak (2003) report that

”problems with correlation have been observed between sequences generated from higher primes”.

Animation

As many scenes are not static but instead change over time, it could be of interest to be able to animate the scene. While some basic animations were included in the rendering tool at an early stage of development, these were later removed to leave time for implementing other features. This is however something that could be added in the future.

Chapter 8

Conclusion

The aim of this project was to create tools and explore solutions that could improve the 3D content creation and rendering process. More specifically, in the context of creating product images of furniture. Three suggested improvements to some of the identified problems have been presented and evaluated in this report.

The modeling tool can successfully be applied to create a chair design featuring interwoven rattan strands. Even though the resulting model looks artificial, it can still bring value and assist the artist during the modeling process by producing much of the base work. With additional parameters and added variation the resulting geometry from the tool could be improved.

The shader for woven cloth produces realistic results for certain types of fabric. The span of materials that the shader is capable of representing could be increased by allowing more freedom when setting parameters for the individual yarns. For the right types of cloth, a specialized woven cloth shader has great benefits over other methods, freeing up memory for other uses and showing visual effects that are not present in other methods.

Finally, a simple interactive ray tracer was implemented using the OptiX framework. Using OptiX has allowed the ray tracing processes to be parallelized on the GPU, successfully accelerating the rendering process. With new software developments and quickly increasing hardware capabilities, the results gathered in this project indicate that ray tracing may be able to be used for real time applications to a larger extent than it is today.

Computer graphics is currently successfully used for producing product images in the industry. Still, there are certain objects and materials that can not be plausibly reproduced through 3D graphics. Another problem is the long feedback loop caused by the slow rendering times of ray tracing. The results of this project show that it is possible to solve some of these issues with focused research. This has the potential to lower the costs and environmental impact of producing photorealistic product images.

References

Adabala, N., Magnenat-Thalmann, N., & Fei, G. (2003). Visualization of woven cloth. In *Proceedings of the 14th eurographics symposium on rendering* (pp. 178–186). The Eurographics Association.

Adanur, S. (2001). *Handbook of weaving.* Boca Raton, Fla: CRC Press. Retrieved from `https://books.google.se/books?id=SshI5QYlgKMC&dq=fundamentals+of+weaving+weft&source=gbs_navlinks_s`

Akeley, K., Kirk, D., Seiler, L., Slusallek, P., & Grantham, B. (2002). When will ray-tracing replace rasterization? In *Acm siggraph 2002 conference abstracts and applications* (pp. 86–87). New York, NY, USA: ACM. doi: 10.1145/1242073.1242120

Akenine, T., Haines, E., & Hoffman, N. (2010). *Real-time rendering* (3rd ed.). Roca Raton, Florida: CRC Press.

Assarsson, U. (2015). *Computer graphics.* Retrieved 2016-05-10, from `http://www.cse.chalmers.se/edu/year/2015/course/TDA361/`

Autodesk. (n.d.). *3ds max.* Retrieved from `http://www.autodesk.com/products/3ds-max/`

Blinn, J. F. (1977). Models of light reflection for computer synthesized pictures. In *Proceedings of the 4th annual conference on computer graphics and interactive techniques* (pp. 192–198). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/563858.563893` doi: 10.1145/563858.563893

Burley, B., & Walt Disney Animation Studios. (2012). Physically-based shading at disney. In *Acm siggraph* (pp. 1–7).

Chaos Software. (n.d.). *V-ray for 3ds max.* Retrieved from `http://www.chaosgroup.com/en/2/vray.html`

ChaosGroup. (2016). *Vray rt.* Retrieved 2016-05-14, from `http://docs.chaosgroup.com/display/VRAY3MAX/About+V-Ray+RT`

Chris 73. (2005). *Rattan chair.* Retrieved from `http://commons.wikimedia.org/wiki/File:Rattan_chair.jpg`

Cirio, G., Lopez-Moreno, J., Miraut, D., & Otaduy, M. A. (2014, November). Yarn-level simulation of woven cloth. *ACM Trans. Graph.*, *33*(6), 207:1–207:11. doi: 10.1145/2661229.2661279

Creative Tools AB. (2011). *Blue cow-horse figure.* Retrieved 2016-04-02, from `http://www.thingiverse.com/thing:7389`

Hanrahan, P., & Krueger, W. (1993). Reflection from layered surfaces due to subsurface scattering. *Proceedings of the 20th annual conference on ...*, 165–174. doi: 10.1145/166117.166139

Heitz, E. (2014). *Understanding the Masking-Shadowing Function in To cite this version* (Tech. Rep. No. February).

Hess, S., & Polak, J. (2003). *An alternative method to the scrambled halton sequence for removing correlation between standard halton sequences in high dimensions.*

Hoffman, N. (2013). Course notes: Background: Physics and Math of Shading. In *Siggraph 2013 courses.* Anaheim.

Irawan, P. (2008). *Appearance of woven cloth* (Unpublished doctoral dissertation). Cornell University.

Irawan, P., & Marschner, S. (2012). Specular reflection from woven cloth. *ACM Transactions on Graphics*, *31*(20), 1–20. doi: 10.1145/2077341.2077352

Iwanaga, M., Hatano, T., Ishihara, T., & Vengurlekar, A. S. (2006). Reciprocal transmittances and reflectances: An elementary proof. *Science And Technology*(1), 6.

Jimenez, J., Echevarria, J. I., Sousa, T., & Gutierrez, D. (2012). *Smaa: Enhanced subpixel morphological antialiasing.*

Kaldor, J. M., James, D. L., & Marschner, S. (2008, August). Simulating knitted cloth at the yarn level. *ACM Trans. Graph.*, *27*(3), 65:1–65:9. doi: 10.1145/1360612.1360664

Khungurn, P., Schroeder, D., Zhao, S., Bala, K., & Marschner, S. (2015). Matching Real Fabrics with Micro-Appearance Models. *ACM Trans. Graph.*, *35*(1), 1:1—-26.

Marschner, S. (2007). CS667 Lecture Notes : Scattering. *Distribution*(August), 1–8.

Marschner, S. R., Jensen, H. W., Cammarano, M., Worley, S., & Hanrahan, P. (2003, July). Light scattering from human hair fibers. *ACM Trans. Graph.*, *22*(3), 780–791. Retrieved from `http://doi.acm.org/10.1145/882262.882345` doi: 10.1145/882262.882345

Niederreiter, H. (2014). Recent constructions of low-discrepancy sequences. *Mathematics and Computers in Simulation*, -. doi: http://dx.doi.org/10.1016/j.matcom.2014.10.001

Nielsen, R. (1997). *Wif weaving information file version 1.* Retrieved 2016-04-08, from `http://www.mhsoft.com/wif/wif.html`

Nvidia Software. (2015a). *nvidia optix.* Retrieved from `https://developer.nvidia.com/optix`

Nvidia Software. (2015b). nvidia optix ray tracing engine programming guide [Computer software manual]. (Version 3.9)

Nvidia Software. (2015c). nvcc :: Cuda toolkit documentation [Computer software manual]. Retrieved from `http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#ixzz46BE8wPEQ` (Version 3.9)

Nvidia Software. (2015d). ptx isa :: Cuda toolkit documentation [Computer software manual]. Retrieved from `http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#ixzz46BE8wPEQ` (Version 3.9)

Parker, S., Martin, W., Sloan, P.-P. J., Shirley, P., Smits, B., & Hansen, C. (1999). *Interactive ray tracing.*

Parkin, K. (2014). *Building 3d with ikea.* Retrieved 2016-02-12, from `http://www.cgsociety.org/index.php/cgsfeatures/cgsfeaturespecial/building_3d_with_ikea`

Pellacini, F. (2016). *Yoctogl.* Retrieved 2016-05-14, from `https://github.com/xelatihy/yocto-gl`

Perlin, K. (2002a). *Improved noise reference implementation.*

Perlin, K. (2002b). Improving noise. *ACM Transactions on Graphics*, *21*(3), 2–3. doi: 10.1145/566654.566636

Pharr, M., & Humphreys, G. (2010). *Physically based rendering: From theory to implementation* (2nd ed.). Burlington, Massachusetts: Morgan Kaufmann.

Piegl, L., & Tiller, W. (1997). *The nurbs book* (2nd ed.). Berlin: Springer-Verlag.

Prautzsch, H., Boehm, W., & Paluszny, M. (2002). *Bézier and b-spline techniques.* Berlin, Heidelberg: Springer Berlin Heidelberg.

Rademacher, P. (2016). *Ray tracing.* Retrieved 2016-05-12, from `https://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html`

Råde, L., & Westergren, B. (2004). *Mathematics handbook beta.* Studentlitteratur AB.

Schröder, K., Zhao, S., & Zinke, A. (2012). Recent Advances in Physically-Based Ap-

pearance Modeling of Cloth. In *Siggraph asia 2012 courses* (pp. 12:1–12:52). doi: 10.1145/2407783.2407795

Sew4Home. (2010). *All about fabric weaves: A tutorial.* Retrieved 2016-03-22, from `http://www.sew4home.com/tips-resources/buying-guide/all-about-fabric-weaves-tutorial`

Styperek, B. (n.d.). *Mitsuba addon for blender.* Retrieved from `https://www.mitsuba-renderer.org/plugins.html`

Suffern, K. (2007). *Ray tracing from the ground up.* Natick, MA, USA: A. K. Peters, Ltd.

Theoretical and Computational Biophysics Group. (2016). *Interactive gpu ray tracing.* Retrieved 2016-05-14, from `http://www.ks.uiuc.edu/Research/vmd/vmd-1.9.2/optix.html`

Wald, & Slusallek. (2001). *State of the art interactive ray tracing.*

Walter, B., Marschner, S. R., Li, H., & Torrance, K. E. (2007). Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th eurographics conference on rendering techniques* (pp. 195–206).

Wang, C., Xie, F., & Krishnamachari, P. (2014). Importance sampling for a microcylinder based cloth bsdf. In *Acm siggraph 2014 talks on - siggraph '14* (pp. 1–1). doi: 10.1145/2614106.2614134

Wheeler, D. J., & Needham, R. M. (1994). TEA, a tiny encryption algorithm. In *Lecture notes in computer science* (pp. 363–366). doi: 10.1007/3-540-60590-8_29

Yang, L. (2009). *CS 348b Final Project : Cloth Rendering* (Tech. Rep.).

Yuksel, C., Kaldor, J. M., James, D. L., & Marschner, S. (2012). Stitch Meshes for Modeling Knitted Clothing with Yarn-level Detail. *ACM Trans. Graph. Article*, *31*(12). doi: 10.1145/2185520.2185533

Zhao, S., Jakob, W., Marschner, S., & Bala, K. (2012). Structure-aware synthesis for predictive woven fabric appearance. In *Siggraph 2012 proceedings* (Vol. 31, pp. 1–10). doi: 10.1145/2185520.2335426

Appendix A

Listings

In this appendix, code segments are presented which can help to demonstrate how certain components have been implemented.

A.1 Listings for the Cloth Shader

A.1.1 API

Listing A.2 shows how the how the general shader is called from a hypothetical rendering engine.

Listing A.1: Example implementation of a plug-in for a rendering engine using the generalized shader.

```
class WovenClothMaterial {
    wcWeaveParameters m_weave_parameters;

    void init(...) {
        //Set global weave params from ui
        m_weave_parameters.uscale = uscale;
        m_weave_parameters.vscale = vscale;
        m_weave_parameters.umax   = umax;
        m_weave_parameters.psi    = psi;
        m_weave_parameters.alpha  = alpha;
        m_weave_parameters.beta   = beta;
        m_weave_parameters.delta_x = delta_x;

        //Load WIF file
        wcWeavePatternFromWIF(&m_weave_parameters,filename);
    }

    Color eval(..., wcWeaveParameters *weave_parameters) {

        //...

        wcIntersectionData intersection_data;
        intersection_data.wi_x = wi.x;
        intersection_data.wi_y = wi.y;
        intersection_data.wi_z = wi.z;
        intersection_data.wo_x = wo.x;
        intersection_data.wo_y = wo.y;
        intersection_data.wo_z = wo.z;
        intersection_data.uv_x = uv.x;
        intersection_data.uv_y = uv.y;

        //Get pattern data at current point
```

```
        wcPatternData pattern_data =
            wcGetPatternData(intersection_data,weave_parameters);

        //Get specular and diffuse contribution
        Color specular(1.f,1.f,1.f);
        specular *=
            wcEvalSpecular(intersection_data,pattern_data,weave_parameters);

        Color diffuse;
        wcEvalDiffuse(intersection_data,pattern_data,weave_parameters,
            &diffuse.r, &diffuse.g, &diffuse.b);

        return specular_strength * specular + (1.f -
            specular_strength)*diffuse;
    }
}
```

A.1.2 Variation

In the cloth shader, variation is achieved by introducing noise. There are two types of variation: *highlight variation* and *yarn variation*. Listing A.2 show how yarn variation is applied. Notice that the yarn variation calls the `octavePerlin` function. This function returns noise that is calculated by superimposing multiple samples of Perlin noise (Perlin, 2002b), which is shown in Listing A.3.

Listing A.2: Function evaluating the smooth variation to be applied at current point.

```
static float yarnVariation(wcPatternData pattern_data,
        const wcWeaveParameters *params)
{
    float variation = 1.f;

    uint32_t tindex_x = pattern_data.total_index_x;
    uint32_t tindex_y = pattern_data.total_index_y;

    //Switch X and Y for warp, so that we have the yarn going along y
    if(!pattern_data.warp_above){
        float tmp = tindex_x;
        tindex_x = tindex_y;
        tindex_y = tmp;
    }

    //We want to vary the intesity along the yarn.
    //For a parallel yarn we want a different variation.
    //Use a large xscale to make the values very different.

    float amplitude = params->yarnvar_amplitude;
    float xscale = params->yarnvar_xscale;
    float yscale = params->yarnvar_yscale;
    int octaves = params->yarnvar_octaves;
    float persistance = params->yarnvar_persistance;

    float x_noise = (tindex_x/(float)params->pattern_width) * xscale;
    float y_noise = (tindex_y + (pattern_data.y/2.f + 0.5))
        /(float)params->pattern_width * yscale;

    variation = octavePerlin(x_noise, y_noise, 0,
            octaves, persistance) * amplitude + 1.f;

    return wcClamp(variation, 0.f, 1.f);
}
```

Listing A.3: Function superimposing multiple values from a Perlin noise function at different frequencies and amplitudes.

```
static double octavePerlin(double x, double y, double z, int octaves,
    double persistance) {
    double total = 0;
    double frequency = 1;
    double amplitude = 1;
    double maxvalue = 0;

    int i;
    for(i = 0; i < octaves; i++) {
        total += noise(x*frequency, y*frequency, z*frequency) * amplitude;

        maxvalue += amplitude;
        amplitude *= persistance;
        frequency *= 2;
    }
    return total/maxvalue;
}
```

A.2 Lisings for the Optix Renderer

A.2.1 Cloth Shader Integration

This algorithm is used by the OptiX CUDA file for reading the cloth material.

Listing A.4: The core part of the material which implements the cloth shader from Chapter 4.

```
        // If not completely shadowed, light the hit point
        if( fmaxf(light_attenuation) > 0.0f ) {
            wcPatternData pattern_data =
                wcGetPatternData(intersection,&params);
            float specular =
                wcEvalSpecular(intersection,pattern_data,&params);%

            float3 Lc = light.color * light_attenuation;

            //The color stored here will then be used to color a pixel's
                ray
            color +=  nDl * Lc * ((1.f - wc_specular_strength) *
                make_float3(pattern_data.color_r, pattern_data.color_g,
                pattern_data.color_b)
                 + wc_specular_strength * specular);
        }
```

A.3 Source Code Access

Some of the source code implemented for this project is available for study in git repositories hosted on GitHub.

Tool for rattan modeling

`https://github.com/karljakoblarsson/Rattan-Geometry`

Cloth shader

`https://github.com/vidarn/cloth-shader`