Technische Universität Wien

Bachelorarbeit

im Studiengang Medieninformatik und Visual Computing

Definition of a Workflow to Import FBX Models in Unity3D at Run-Time While Retaining Material Properties for Various Shader Types

---

Author
Nicolas Themmer
0929245

Advisor
Michael Wimmer

Assistance
Johannes
Unterguggenberger

December 1, 2018

Acknowledgements

I take this opportunity to express gratitude to my supervisor for his help and support. I also thank my parents for their encouragement and support. I am also grateful to my partner who supported me throughout this venture.

Abstrakt

Die vorliegende Bachelorarbeit lässt sich in zwei aufbauende Teile unterteilen. Zum einen werden physikalisch basierende Shader untersucht und ihre Funktionsweise analysiert. Im Zuge dessen werden Shader im Allgemeinen studiert, wobei wichtige Elemente wie die Rendering-Gleichung sowie die Grafikpipeline besprochen werden. Ein genaueres Augenmerk wird anschließend auf physikalisch basierende Shader gelegt. In diesem Schritt beschäftigt sich die Arbeit mit unterschiedlichen Phänomenen, die in der Realität auftreten und in der Computergraphik umgesetzt werden konnten. Dabei wird die BRDF (*Bidirectional Reflection Distribution Function*) theoretisch erklärt und mathematisch analysiert. Zum anderen behandelt die Arbeit den Import von dem FBX Dateien in die Game-Engine Unity3D. Das Ziel in diesem Teil des Projektes ist es einen Import zu ermöglichen durch welchen keine Materialinformationen verloren gehen. Im Normalfall müssen nach dem Import von FBX Dateien, Texturen dieser Modelle manuell auf Unity Materialien zugeornet werden. Im Rahmen dessen wird die Autodesk API verwendet um FBX Dateien zu untersuchen und Informationen zu Texturen und Materialien zu erhalten. Einerseits behandelt die Arbeit den Use-Case, der den Upload der FBX Dateien auf einen Server beschreibt, welcher die Dateien analysiert, ein Unity eigenes Format (AssetBundle) mit richtigen Materialinformationen generiert und anschließend in einer Datenbank persistiert. Andererseits wird ein Client implementiert, welcher sich zur Laufzeit mit der Datenbank synchronisiert und Modelle mit korrekten Materialinformationen in die Szene lädt. Dafür wird außerdem die Modellierung von 3D Objekten behandelt und wie Materialien mit zugehörigen Texturen erstellt werden können. Das Projekt ist als Netzwerkapplikation implementiert um den Konvertierungsprozess auszulagern, wodurch die Rechenleistung erheblich reduziert werden kann.
Die Arbeit definiert notwendige Schritte um eine automatische Zuordnung von Texturen auf Unity Materialien zu ermöglichen. Hiermit wird eine Lösung für den Use-Case vorgestellt, wodurch FBX Dateien zur Laufzeit in Unity Applikationen eingebunden werden können.

Abstract

This bachelor thesis can be divided into two sequential parts. The first part examines physically-based shader analyses their functionality. In this context, shader were studied in general, while discussing core elements like the rendering equation and the graphics pipeline. Physically-based shader were subsequently brought to close attention. In this step, the study deals with various real life phenomenons, that occur in reality and were successfully implemented in computer graphics. For this purpose, the BRDF (Bidirectional Reflection Distribution Function) was explained theoretically and analysed mathematically. The second part of this thesis covers the import of FBX files into the game engine Unity3D. The goal of this chapter is to modify the import process to the extend that material information won't get lost. When importing FBX files into Unity3D, textures have to be assigned to Unity materials by default. Therefore, the Autodesk API is used to examine FBX files and gain necessary information regarding textures and materials. The thesis covers the use-case of uploading FBX files to a server, analysing these files, generating Unity files in a native format (AssetBundles) with correct material information and storing them into a database. A client, that synchronises itself with the database during run-time and loading these models into a visual scene was also implemented. In this context, the process of modeling 3D objects, including materials and textures is covered as well. The workflow was implemented as a network application in order to outsource the conversion process and therefore substantially decrease the consumption of computational power.
The workflow defines necessary steps, to automate the assignment of textures to Unity materials. A solution for the use-case of including FBX files into unity applications during run-time is hereby presented.

Contents

1 Introduction

Shader programs are essential for any visual presentation in rendering applications and have been developed a lot over the past years. While the industry strives to make virtual scenes as realistic as possible, different methods emerged to reach that goal. As the name already suggests, physically-based shaders are trying to take various physical variables from the real world into account. Since physically-based shaders became extremely popular, many 3D computer graphics programs implemented support for these shaders.
In game development, 3D objects are created with modeling software and are then imported into game engines. The Autodesk FBX format is commonly used for the import and export process. This format can be loaded into the 3D game engine Unity3D (or briefly Unity), but textures have to be assigned to the object's materials manually. Automating this process makes it easier for developers to work with Unity and FBX files. Furthermore, it opens up the possibility to include FBX files into running applications without a pre-processing step. The theoretical part of this thesis will explore the inner workings of shaders in general and examine physically-based shaders in particular. The goal of the practical part is to automate the process of converting FBX files into Unity AssetBundles in order to import them into running Unity Clients, while upholding correct visual representations. AssetBundles are archive files containing Unity specific components and were specifically created for the run-time import. The main contribution of this thesis is a description of a workflow, that enables the import of FBX models into Unity at run-time while retaining material properties of the original models during the import and making them available in Unity to be directly usable with, e.g., Unity's *Standard Shader*. This is not possible with Unity's own importer, which loses material information during the import process. Another contribution is the description of that workflow in a distributed system, which offers following advantages to a local-only workflow: If FBX models are loaded into multiple Unity clients, computational power can be saved by converting the model only once and storing the AssetBundle in a database. Without local conversion, Unity clients can be synchronised quickly and with little computational costs. Chapter 2 and

3 of this thesis describe necessary components and the theory behind the implementation physically-based shader. Chapter 4 presents the workflow and how physically-based shader can be implemented in Unity.

2 What is a shader?

To understand physically-based shaders (PBS), it is necessary to describe shaders in general and how they are being used. Shaders can be explained as code that runs on GPUs, creating a visual representation of an object's surface. In order to achieve this goal, the shader has to simulate the interaction of light on a surface at a microscopic level. While there are also shaders that deliberately create an unrealistic visual appearance of objects, most shaders take a realistic approach as a priority [6].

2.1 The physical world

When shader programs take a realistic approach, our perception of the physical world must be analysed to get a better understanding of which variables have to be taken into account. Therefore, we need to recognize why visual systems record objects and reflections the way they do. Three different materials are presented in figure 1. The human brain has no problem identifying those materials because light behaves differently on fiber compared to wood, rust or any other material [6].

Figure 1: Fiber, rust and wood material

In the physical world, light is seen as a wave or as particles. Surfaces of objects fundamentally consist out of atoms, which the human eye is not able to perceive. Surfaces that appear to be smooth can be quite rough at a microscopic level, which makes the light scatter in many directions. The way light rays behave after hitting an object's surface determines the

appearance of a material. It can either be absorbed, reflected, refracted or scattered. More on the behaviour of light can be read in chapter 2.3. Calculating surfaces and the resulting reflections at a microscopic level would be too costly and would require unrealistic computational power. 3D models, created by 3D artists, consist of vertices, which are connected to form triangles or quadrilateral ("quads"), see figure 2 [6].

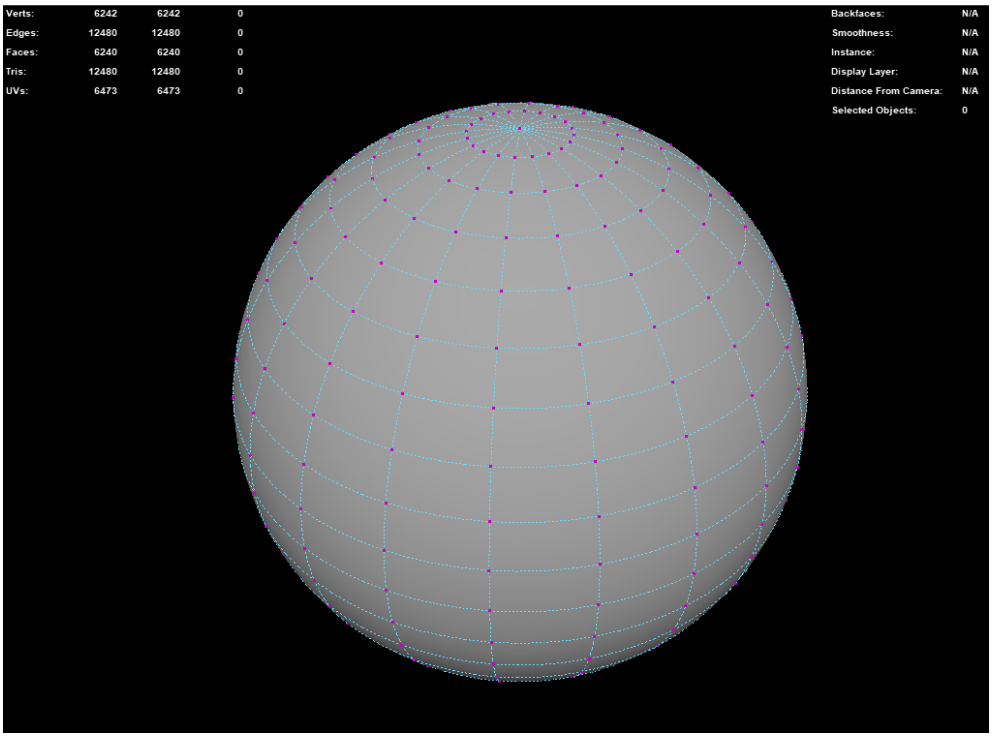

Figure 2: Sphere vertices

A basic 3D object can have a lot of vertices. The sphere in figure 2 has 12480 triangles, which shows how quickly computational resources can be occupied. Rendering at a microscopic level to map the real world is unrealistic. Shaders use a mathematical approach instead to approximate different lighting behaviours on various surfaces [6].

2.2 Shader Types

There are a few types of shaders used by modern graphic cards for the rendering process:

- Vertex Shader: This shader is executed on every vertex and is a well established and common kind of shader.
- Tesselation Shader: Takes data from the vertex shader and is able to create new vertices in the geometry by interpolating original ones [5].
- Geometry Shader: Can change or expand the given geometry by creating new vertex groups and vertices [5].
- Fragment Shader: Is also known as *Pixel Shader* and is executed for each possible final pixel (also known as *fragment*). Fragment Shaders are commonly executed on fragments to determine the color of the corresponding pixel [5].
- Compute Shader: Are useful for calculations that can be broken down into a large amount of independent tasks because they are able control how threads are being handled in a given shader invocation. Threads have read/write permissions to a common memory pool, which allows them to access and share calculations [8][6].

2.3 Lighting

Light sources, such as light bulbs, have to be created with light-emitting surfaces. Other surfaces will reflect the incoming light and will therefore be illuminated. Light rays will be scattered from one surface to another, which can be traced back in a recursive manner [7].

Referring to figure 3, the initial light source, represented by the sun, and the light rays, hitting the surfaces A and B, can be seen. Both surfaces reflect parts of the incoming light back to the other surface. The light will be reflected again, creating a recursion. This recursive process can mathematically be described by using the **rendering equation**.

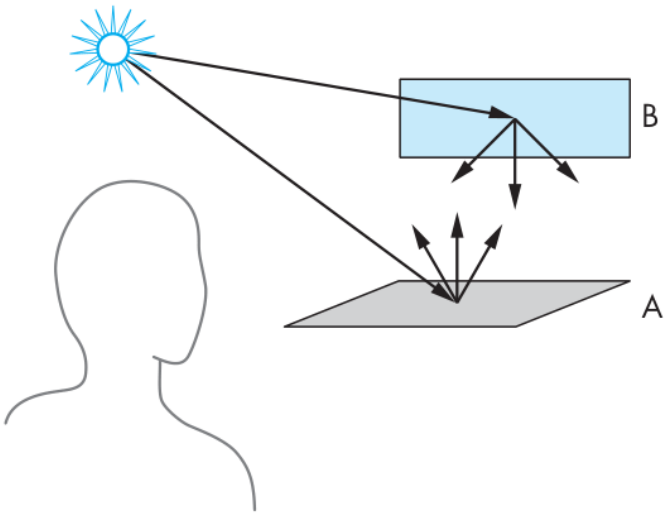

Figure 3: Reflecting surfaces (reprinted from [7]).

This integral equation summarizes all parameters to calculate lighting and cannot be solved analytically but various approaches, such as ray tracing, can be used to approximate the equation [7].

$$L_0(x, w_0) = L_\varepsilon(x, w_0) + \int_\Omega f(x, w_i \rightarrow w_0) \cdot L_i(x, w_i) \cdot (w_i \cdot n) dw$$

- The term $L_0(x, w_0)$ is retrieved from the equation and describes the outgoing light in direction w_0 at position x [6].
- $L_\varepsilon(x, w_0)$ is added once and describes the emitted light in direction w_0 from the surface point x [6].
- The following integral $\int_\Omega f[...]dw$ is a repeated summation for all solid angles in the corresponding hemisphere Ω above point x [6].
- The term $f(x, w_i \rightarrow w_0)$ within the hemisphere represents the BRDF and defines the proportion of light reflected from w_i (negative direction of the incoming light) to w_0 at position x (see chapter 3.3) [6].
- $L_i(x, w_i)$ describes the incoming light onto the surface point x [6].

- The term $(w_i \cdot n)$ specifies the normal attenuation, which attenuates the incoming light at x based on the angle between the normal vector n and the incoming light direction w_i [7].

Reflected light rays end up hitting other surfaces, that will reflect parts of the light rays back again. Light rays will keep bouncing around until their energy is spent. We therefore differentiate between:

- Direct light: Light hitting a surface directly.
- Indirect light: Light hitting a surface after being reflected from another surface.

2.4 The Graphics Pipeline

When dealing with shaders it is necessary to understand the basic concepts of the rendering pipeline. This chapter gives a general overview of how 3D scenes are being rendered. There are a few widely used graphics APIs, that give developers the option to render a 3D scene. APIs like OpenGL take advantage of hardware acceleration and are mainly optimized to run on GPUs [7]. The graphics pipeline can be broken down and simplified in following steps (see 4) [6]:

1. In the first stage, all the information has to be gathered and organized, in order to be used in the following steps. The information contains data from the scene, which contains meshes, textures and materials [6].
2. With the data information at hand, the vertex processing stage runs the *Vertex Shader* on each vertex in the scene [6].
3. The *Vertex Post-Processing* stage removes 3D data, which are not in the view frustum and will therefore not be seen on the screen. This process is referred to as *clipping* and is an essential step for saving computational power [6].
4. The primitive assembly stage prepares the input received from the previous stage and sends it to the next one [6].

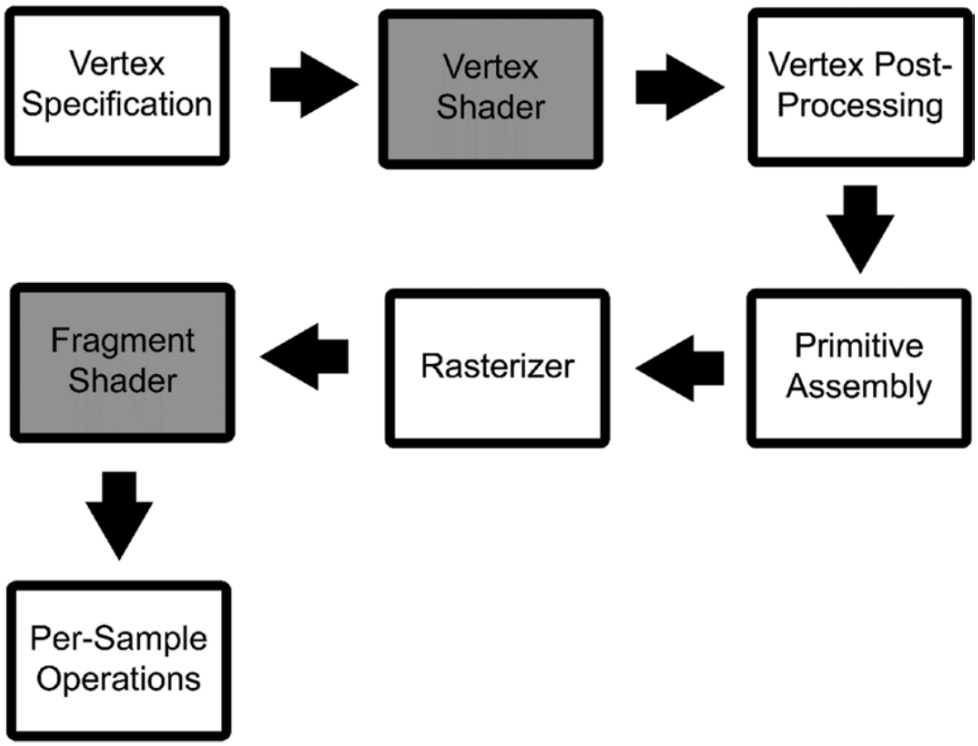

Figure 4: Graphics Pipeline (reprinted from [6])

5. The rasterizer is part of the GPU. It takes a triangle and calculates the corresponding pixels (fragments) on the final image. Since all three vertices from the triangle contain attributes, the rasterizer produces an interpolated output for the fragments based on the vertex data [6].

6. The fragment shader stage takes all the fragments from the previous step and runs the fragment shader on each of them [6].

The geometry and the tesselation shader, mentioned in chapter 2.2, would be located between the vertex and fragment shader in the graph above. Physically-based shader can generally be created by implementing vertex and fragment shader.

3 Physically-Based Shading

Physically-based shading is a common shading technique. The main difference to other shading calculations is that the detailed behaviour of light is taken into account, when reflections and other light phenomena are calculated. The physics of light and how light interacts with different kinds of materials is essential for the understanding of PBS [6].

3.1 Light-material interactions

When a ray of light strikes the surface of an object, some parts of it will be reflected, some absorbed and some will be refracted. The understanding of how much a ray behaves in either of these three ways, is essential for making shaders as realistic as possible.

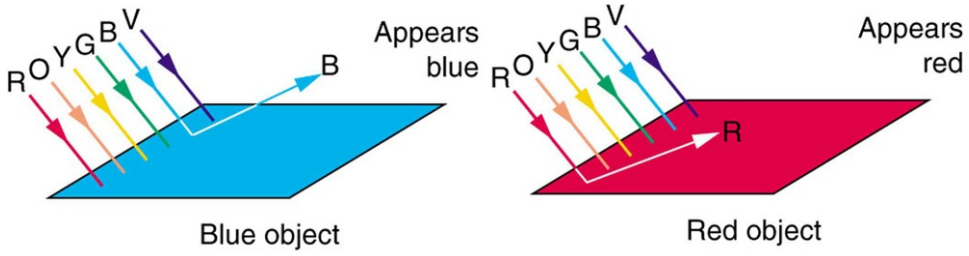

Figure 5: Rays of light hitting a blue and a red object (reprinted from [11]).

- The process of *absorption* determines the true color of an object. Light is essentially composed of different wave lengths, which we perceive as different colors. Absorption eliminates all the colors in that range, except the one that can be seen on the object's surface (see figure 5).
- *Reflection* focuses on the part of the light ray, that is being reflected (not absorbed) from the object. For PBS, not only the direction of the reflection is important but also the amount of the light ray's energy used by reflectance.
  Every object has a different surface at a microscopic level but

3D models are not created to an extent that is able to represent such a level of detail. Therefore, these surface details have to be simulated. Furthermore, PBS will not only calculate one direction of reflectance for every pixel but will reflect the light in multiple directions. Directions are calculated statistically and result in a cone-like form (see figure 6) [6]. The reflected light is dependent on

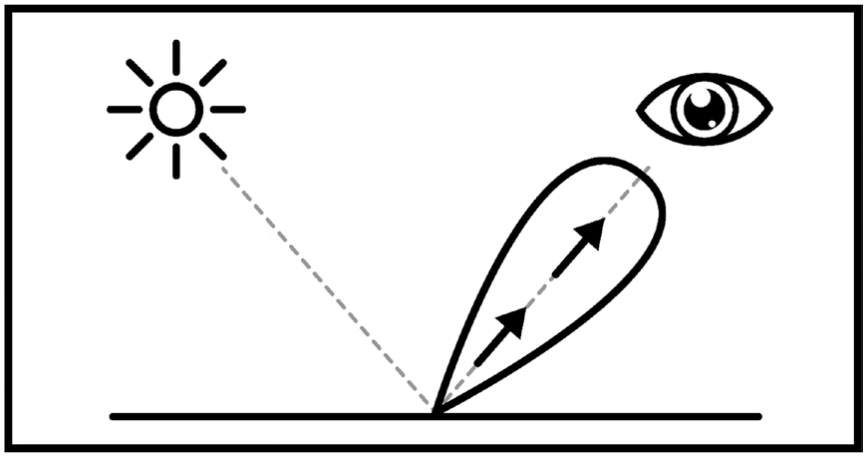

Figure 6: Reflection of rays into a cone (or lobe) (reprinted from [6]).

the incoming light which is typically calculated over a hemisphere, which is placed on top of the pixel. Since light can be conceived from multiple directions at once, all rays within the hemisphere need to be added up to create a realistic representation of light conditions [6].

- If light hits a semitransparent object, such as glass or water, it digs through and hits the layer below. This process is called *transmission* or *refraction*. Materials are created with a refraction index, that represents the level of transparency [6].

Light rays are either direct or indirect (see chapter 2.3). Calculating indirect light is extremely expensive and needs to be approximated through different techniques like global illumination, light mapping and reflection probes for reflective materials [6].

3.2 Fresnel Reflectance

A simple specular approximation does not limit the specular intensity in regard to the incoming light. To make shader physically-based, the energy of outgoing light has to be limited by the energy of incoming light. The amount of reflected and refracted light is dependent on the incoming light angle, the normal vector and the viewing angle [6].

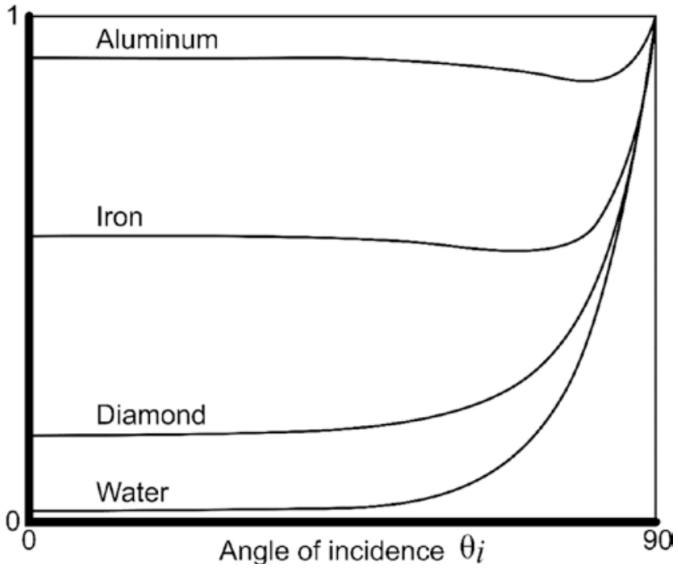

Figure 7: Fresnel Reflectance (y-axis) for differenct materials in regard to the angle of incidence (reprinted from [6]).

Fresnel Reflectance describes a realistic physical phenomenon: On the one hand being able to see underneath, when standing above the water and looking straight down. On the other hand not being able to see underneath when gazing into the horizon due to reflection. From the angle of incidence between 0 and 45 degrees most materials react the same way. From 45 to 90 degrees, reflectance gradually increases (see figure 7). The Fresnel equation is used to calculate the ratio between reflection and refraction and is usually implemented in code through the *Schlick approximation*, which has a similar behaviour to physical world phenomenons. F_0 describes the specular color, that can be seen at a 0° angle of incidence [6].

$$F_{schlick}(F_0, l, h) = F_0 + (1 - F_0)(1 - (l \cdot h))^5$$

l represents the incoming light direction and h stands for the surface's normal vector. When using the normal vector in the microfacet BRDFs, the h vector is used instead (see chapter 3.5) [10]. Fresnel reflection can be proven by performing a simple experiment: While sitting in front of a monitor, hold a smartphone close to the stomach. Tilt it until it reflects the monitor. The reflection will be weak due to the low angle of incidence. The same experiment can be repeated, while holding the smartphone in front of the eyes. The reflection intensity will be a much greater than before, as the angle of incidence increased substantially [1].

3.3 BRDF

The BRDF (bidirectional reflection distribution function) is a function used to calculate the intensity of reflectance on an object. The intensity of reflected light depends on the incoming light direction l, that reflects a specific amount of light towards the outgoing direction v (see figure 8) [1].

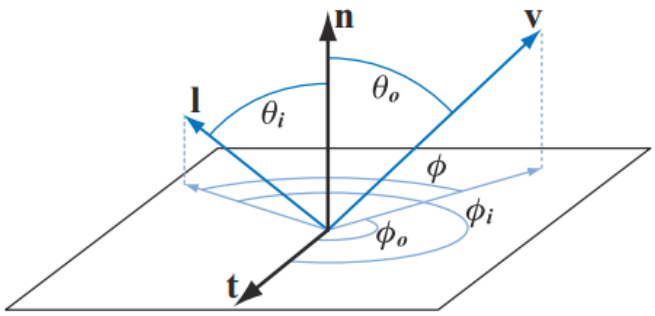

Figure 8: The BRDF. ϕ_o and ϕ_i as azimuth angles with respect to tangent vector t. θ_o and θ_i as elevation angles with respect to the normal n (reprinted from [1]).

The BRDF is *bidirectional* because the functions depends on two directions. These directions are given with respect to the surface. Each

vector is therefore described by two numbers: the altitude angle (elevation) and the azimuth (horizontal rotation). The altitude angle θ_o stands for the angle between the normal and v, θ_i for the angle between the normal and l. The azimuth angles describe the angle between the incoming or outgoing light direction and the tangent vector t [1].

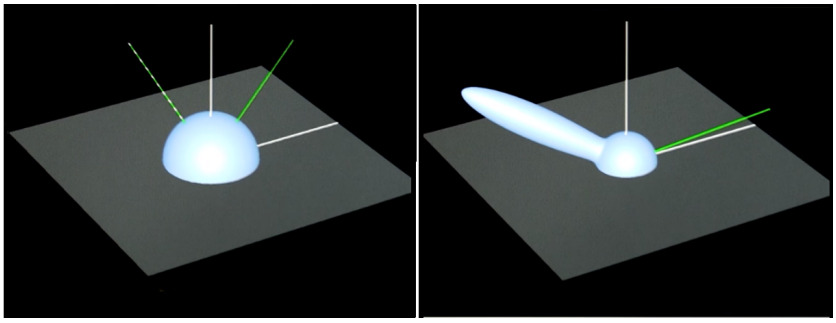

Figure 9: BRDF reflections distribution examples. Left: diffuse, right: specular (adapted from [1])

The term *reflectance distribution* stands for the spreading of light. A widely used distribution is the diffuse reflection (see figure 9). The viewing direction is irrelevant to some incoming light directions. This is what defines diffuse reflection. Consequently, the reflection is represented by the surface of a hemisphere. Specular highlights can be visualized by specular lobes (see figure 9). This distribution produces a glossy surface, where light is reflected in a general direction. The light's direction determines where most of the light's energy is reflected. When the lobe expands, the specular reflection spreads out. The BRDF is an essential part for calculating the outgoing light and is used in the rendering equation (see chapter 2.3) [1].

3.4 Microgeometry

Surface irregularities that are smaller than a pixel and are unvisible to the naked eye, are impossible to be modeled explicitly and will therefore be modeled statistically by the BRDF. Surface points contain a lot of microsurface normals that scatter and reflect the light in multiple directions. Since the orientation of these normals are somewhat random,

they can be interpreted as a statistical distribution. The distribution is continuous for most surfaces and peaks at the macroscopic surface normal. The *tighter* the distribution is, the *smoother* the surface will appear [1].

3.5 Microfacet Theory

Most BRDF models are based on the observation how microgeometry affects reflectance. This mathematical analysis is referred to as the *Microfacet Theory*. It is based on the modeling of microgeometry through microfacets, which are small flat facets with a normal vector m. These facets reflect light in regard to the micro-BRDF $f_\mu(l, v, m)$. The combined reflectance of all microfacets will add up to the overall surface BRDF. When ignoring nanoscale irregularities, each microfacet is viewed as a perfect Fresnel mirror and will therefore reflect each incoming ray of light into one outgoing direction. That direction depends on the light direction and the microfacet normal [1].

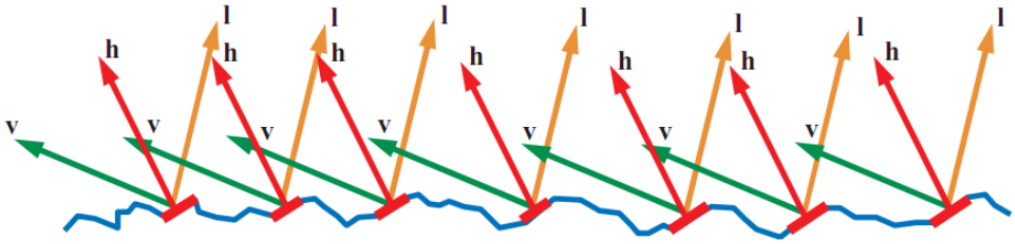

Figure 10: Half vectors defined by the incoming light and view direction. Highlighted red microfacets are visible (reprinted from [10]).

Only the microfacets with their normals m oriented in exactly the same direction as the half vector h reflect any visible light. Any microfacet aiming in a slightly different direction will be bouncing l into some direction other than v and will therefore not influence the BRDF [1]. Not all microfacets where $m = h$ will contribute because some will be blocked by other microfacets. In this case we differentiate between two phenomenons that occur in the Microfacet Theory. Some facets

can either be blocked from the light direction, *Shadowing*, or the view direction, *Masking*. In reality, blocked light will continue to bounce and some of it will eventually contribute to the BRDF. Microfacet BRDFs commonly ignore this fact and assume that all blocked light is lost (see figure 11) [1].

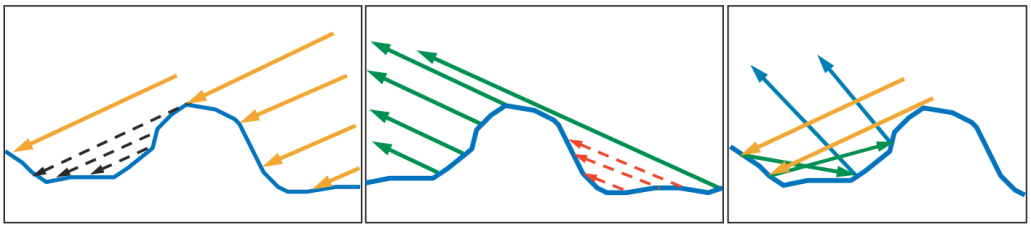

Figure 11: Non visible microfacets due to shadowing (left) and masking (center). Realistic representation of bounced light (right) (reprinted from [10]).

From these assumptions it is possible to derive an equation that describes a basic *Microfacet Specular BRDF*:

$$f(l,v) = \frac{F(l,h)G(l,v,h)D(h)}{4(n \cdot l)(n \cdot v)}$$

$F(l,h)$ describes the *Fresnel* reflectance, covered in chapter 3.2. It represents the fraction of incoming light, that is reflected and not refracted. It varies based on the lighting angle and the surface normal [10].

$G(l,v,h)$ refers to the *Geometry Function*, that explains how many of the visible microfacets will not be rendered due to shadowing or masking. *Smith's function* is often used as the geometry function, which is both mathematically valid and physically realistic [10].

$D(h)$ refers to the distribution of microfacets pointing to the direction where $h = m$ (see chapter 3.4) [10].

Understanding how light bounces on a microscopic level is an essential part for implementing PBS. The microfacet theory helps to simulate this behaviour and must be included for lighting calculations.

4 Workflow Definition

This chapter will demonstrate a possible workflow to automate the import process for FBX models in Unity3D at run-time while retaining material properties. This project gives an example of how specific texture mappings can be initiated in form of a network application. This workflow will not only cover the run-time import of Unity's AssetBundles, but will also cover the modeling process and how the Autodesk FBX format can be parsed with the Autodesk API to extract material information that can later be used to automatically map corresponding textures onto Unity materials. When instantiating Unity materials, the Unity *"Standard Shader"* is applied by default. This shader is physically-based and will be discussed in chapter 4.2.2.

4.1 Introduction

When creating applications involving game engines, models are usually included in the correct format and only have to be included in the application. This workflow not only requires communication between artists and game engine specialists, but also presumes a pre-processing step, where textures have to be dragged onto the model's material. This chapter will present a possible workflow to integrate FBX models into running Unity applications, without requiring any human-computer interaction for assigning textures to materials. The process is divided in the following steps:

1. Asset Creation: The generation of three dimensional models, which includes the creation of geometry and material properties.

2. FBX to Assetbundle: Reading necessary information from FBX files through the Autodesk API and converting the model into AssetBundles.

3. Upload architecture: Explains which services have to be running and how they communicate.

4. Unity import: How the model should be imported into Unity3D.

4.2 Asset Creation

There are many 3D computer graphics programs that can be used for asset creation. For the creation of the assets used in the context of this thesis Autodesk's 3D computer graphics application *Maya* [4] but other common software packages like *Cinema4D, 3ds Max, Blender, MODO, etc.* contain the same features relevant for asset creation. Even though the FBX format is standardized, structural elements can vary. This thesis will use parameters and values listed in the FBX format structure, created by Maya. For the purpose of this project a model of an office space environment was created. This model can be seen in figure 12.

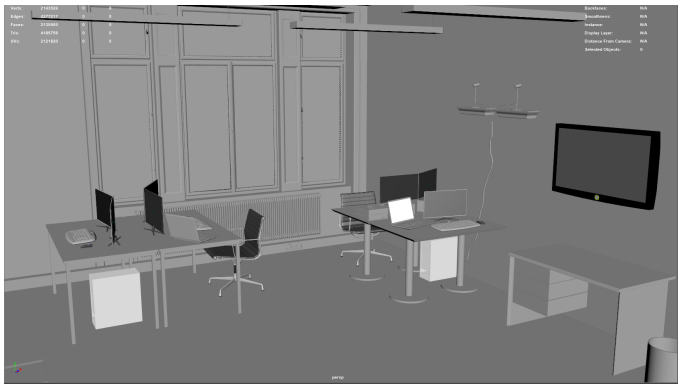

Figure 12: Office space environment without textures, modeled in Maya [4]

4.2.1 UV Mapping

The model in figure 12 is grey because there are no materials and textures added to the elements. In order to assign textures, UV coordinates of each object have to be created. It's necessary to create the geometry of the model before applying UV layouts. When the geometry has changed, UV layouts have to be recreated. UVs are coordinates that describe how 2D textures can be wrapped around 3D objects. UV mapping describes the process of *unfolding* a three-dimensional object into a two-dimensional representation. The UV coordinates specify the locations on these two-dimensional representations and can be used to

describe which point on a texture is assigned to which location on the model [12].

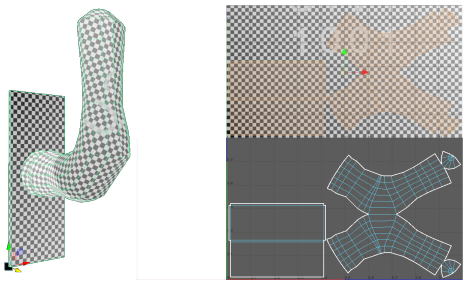

Figure 13: Knob, UV layout with checker board

Figure 13 shows a knob, located on the windows of the office space model (figure 12). Stretched surfaces, caused by UV mapping, should be prevented when unfolding a mesh. The black and white texture (checker board) in figure 13 is used as a reference to understand how the texture is mapped on the surface of the mesh [12]. Ideally all squares should be the same size. Maya [4] offers a large toolset in the UV editor, which makes it easier to achieve the desired result.

4.2.2 Materials and Textures

In the next step, materials can be created. This can either be done with *Maya* or with other 3D painting software like *Substance Painter*[2], that is able to apply materials and add textures to your mesh.

Substance Painter[2] provides users with a set of default materials, that can be altered in any way. The software has the advantage of *preset* materials, which makes it easy to paint details like screws (see figure 14) onto meshes. When texturing a mesh, the material of the object will determine which textures can be applied. Surfaces of objects in the real world contain various properties, that have to be simulated to achieve a realistic visual representation. Unity's standard shader is physically-based and provides the following properties:

- Albedo: Describes the color of diffused light. In other words the raw color of the material itself. This property is the base of the

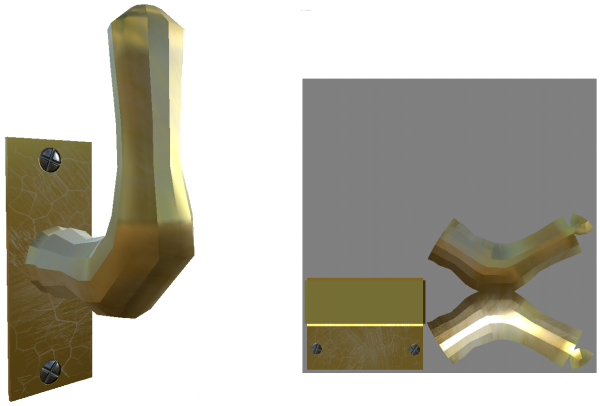

Figure 14: Knob, textured in Substance Painter

material. Either a single color, or a texture map can be chosen. The alpha channel can represent additional information, like the level of transparency.

- Metallic: Defines the metallicness of the material. When the level of metallicness increases, the albedo becomes more and more obscured by the reflections of the envirnonment.
- Smoothness: The smoothness property lets you adjust how smooth or rough the microsurface is. When the microsurface is rough, it will scatter light more easily. Smooth surfaces tend to look glossy, rough surfaces appear matt. A metallic texture map controls the metallicness through the red channel and smoothness through the alpha channel.
- Normal Map: This property adds surface detail to the model without adding geometry. This is used to simulate details like bumps and scratches by changing the way light reflects off the surface.
- Height Map: These maps are similar to normal maps but the rendering technique is more complex and therefore more intensive. Height maps are generally used together with normal maps to show larger changes in surface level.

- Occlusion: This property is a way to add ambient occlusion using a black and white texture. It's a cheap way of adding detail shadows in places where light might have a hard time escaping.
- Emission: This property can be used to make the object emit light. An emission map specifies, which parts of the object are emissive.

When creating materials through *Substance Painter* [2], channels are specified to describe the properties. These channels will be exported in the form of textures.

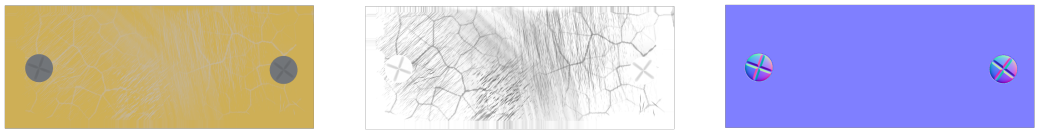

Figure 15: Exported textures from the metallic plate shown in figure 14.

Figure 15 shows the part of the UV coordinate space, that represents the plate behind the knob from figure 14. It shows three different kinds of textures (from left to right):

1. Albedo: The RGBa channel of the mesh. Since the object does not contain any transparent parts, the alpha channel throughout the texture is 0.
2. Metallic: This gray scale texture map shows which parts of the object are metallic and which aren't.
3. Normal Map: Maps that help create more tactile-looking surfaces [12]. In this case, both screws will seem to be elevated.

The process of UV mapping and texturing has to be done for every mesh in figure 12. The final result is displayed in figure 16.

Figure 16: Textured office space environment

4.3 Upload

The main goal of this workflow is to automate the process of converting FBX files into Unity AssetBundles and importing them into a running Unity application. Although this application can be used locally, the project was implemented as a network application, where FBX files will be uploaded from one computer and loaded into another, running the Unity application. The AssetBundle format was specifically created by Unity for run-time import. While it's possible to execute the conversion locally, it's recommended to outsource the process to save computational power. This involves four main steps:

1. Uploading the desired asset in FBX format
2. Convert the FBX file into an AssetBundle and send it to the Backend System
3. Store AssetBundle in the Backend System

4. Synchronise Unity client with the Backend System

The following chapter will focus on the first two points of the uploading service.

4.4 FBX to AssetBundle

To convert FBX files to AssetBundles, incoming FBX files will be received over the network. The following steps will be executed to successfully make the conversion:

1. Building a REST Web Service.
2. Reading material information from FBX files and storing the data in JSON format.
3. Loading FBX files into a Unity scene and applying all texture maps to the Unity Standard Shader, with the help of the previously retained JSON file.
4. Converting the Unity object into an AssetBundle and sending it to the Backend Service

4.4.1 REST Web Service

Since the Autodesk API is written in C++ (see chapter 4.4.2), the REST Service was implemented in C++ as well. This makes the validation of incoming files easier. For the purpose of this project, the C++ REST SDK *Casablanca* [9] was used, which is a Microsoft project for cloud-based client-server communication. Since FBX files will only be uploaded, the process is only able to handle **POST** requests. To send data to the Backend, a client for outgoing connections was created. The program was then deployed on a server to create web service. Creating the REST interface was the first step and the next section will explain how FBX files can be parsed with the Autodesk API.

4.4.2 Parsing FBX files with Autodesk API

The Autodesk API can be used to read information from FBX files. When downloading the Autodesk FBX SDK, a variety of examples for different use-cases can be examined and the corresponding documentation can be found at [3]. The focus of this project resides on extracting material and texture information and packing this data into a JSON string for easy access. FBX files are read as a *scene graph*, which is a tree of **FbxNode** objects. NURBS, lights, cameras and other elements are associated with these nodes but only the mesh element is relevant for this project. To analyse all meshes in a FBX file, the root element has to be read from the scene. This way, all siblings can be parsed.

```
1  void DisplayContent(FbxScene* pScene)
2  {
3     int i;
4     FbxNode* lNode = pScene->GetRootNode();
5
6     if(lNode)
7     {
8      mesh_info.append("[");
9       for(i = 0; i < lNode->GetChildCount(); i++)
10      {
11         DisplayContent(lNode->GetChild(i));
12      }
13    }
14 }
15
16 void DisplayContent(FbxNode* pNode)
17 {
18   FbxNodeAttribute::EType lAttributeType;
19   int i;
20
21   if (pNode->GetNodeAttribute() == NULL) {
22     FBXSDK_printf("NULL Node Attribute\n\n");
23   }
24   else {
25     lAttributeType =
```

```
        (pNode->GetNodeAttribute()->GetAttributeType());
26
27      if (lAttributeType == FbxNodeAttribute::eMesh) {
28        DisplayMesh(pNode);
29      }
30    }
31
32    for (i = 0; i < pNode->GetChildCount(); i++) {
33      DisplayContent(pNode->GetChild(i));
34    }
35  }
```

The code above iterates through all **FbxNode** elements (line number 9) and calls the **DisplayMesh(FbxNode* pNode)** method for each node (line number 28). This function contains two other methods, invoked one after another. The Autodesk API provides the **DisplayMaterial(FbxGeometry* pGeometry)** function, that reads information of the material, which is attached to the mesh. The **DisplayTexture(FbxGeometry* pGeometry)** function outputs the texture type and the corresponding material (see line 12 and 15 below). The information is stored in a JSON string for later usage.

```
1  [{
2    "mesh": "MySphere",
3    "materials": [{
4      "Name": "MyLambertMaterial",
5      "Ambient": "0.500000 (red), 0.500000 (green), 0.500000
           (blue)",
6      "Diffuse": "1.000000 (red), 1.000000 (green), 1.000000
           (blue)",
7      "Emissive": "0.000000 (red), 0.000000 (green), 0.000000
           (blue)",
8      "Opacity": "0.000000",
9      "ShadingModel": "Lambert"
10   }],
11   "textures": [{
12     "Materialname": "MyLambertMaterial",
13     "DiffuseColor": "rock_vstreaks_Base_Color.png"
```

```
  },{
    "Materialname": "MyLambertMaterial",
    "Bump": "rock_vstreaks_Normal-unity.png"
  }]
}]
```

The code above extracts information from FBX files and is built as a *Dynamic Linked Library,* or briefly DLL, with external methods. This way it can be used by Unity as a plugin. The following code describes external C++ methods, that will eventually be called by Unity scripts.

```
EXTERNMETHOD SharedAPI* CreateSharedAPI(int ID) {
  return new SharedAPI(ID);
}

EXTERNMETHOD const char* GetFileInfo(SharedAPI* sharedAPI, const
    char *file) {
  return sharedAPI->GetFileInfo(file);
}

EXTERNMETHOD void DeleteSharedAPI(SharedAPI* sharedAPI) {
  delete sharedAPI;
}
```

4.4.3 Converting FBX file to Assetbundle

Using the script from the previous section, FBX files can now be parsed directly in Unity, while texture to material mappings are stored in a JSON string. Therefore the DLL was imported into the Unity Project's "Plugin" folder. The external methods can be used in Unity by establishing handles to the functions inside the DLL using **DLLImport** statements to make them available in C#.

```
[DllImport("ImportScene")]
private static extern IntPtr CreateSharedAPI(int ID);

[DllImport("ImportScene", CallingConvention =
    CallingConvention.Cdecl)]
```

```
5  private static extern IntPtr GetFileInfo(IntPtr api, string
       file);
6
7  [DllImport("ImportScene")]
8  private static extern void DeleteSharedAPI(IntPtr api);
```

After creating a pointer to the shared API, which is defined through the DLL plugin, the JSON-info string can be received with the **GetFileInfo** method. To initialize meshes with a JSON string, a *serializable* class was added to the project:

```
1  [Serializable]
2  public class Mesh
3  {
4    [JsonProperty("mesh")] public string mesh { get; set; }
5     [JsonProperty("materials")] public List<MaterialData>
         materials { get; set; }
6  }
7
8  [Serializable]
9  public class MaterialData
10 {
11     public string Name;
12     public string Ambient;
13     public string Diffuse;
14     public string Specular;
15     public string Emissive;
16     public string Opacity;
17     ...
18
19     private TextureData Textures;
20 }
21
22 [Serializable]
23 public class TextureData
24 {
25     public string Materialname;
26     public string DiffuseColor;
```

```
    public string SpecularColor;
    public string Bump;
    ...
}
```

After initializing the mesh class, all necessary information for binding textures to specific material slots is available. The following code snippet shows how textures can be applied to corresponding materials, which were obtained by importing the file with Unity's **AssetImporter**.

```
var materials = AssetDatabase
  .LoadAllAssetsAtPath(assetImporter.assetPath)
  .Where(x => x.GetType() == typeof(Material));

foreach (var material in materials)
{
  var newAssetPath = string.Join(PATH,
  new[] {materialsPath, material.name}) + ".mat";

  AssetDatabase.ExtractAsset(material, newAssetPath);
}

TextureData textures = materialData.getTexture();

Texture2D bump = FileHandler.GetTextureByName(textures.Bump);
Texture2D diffuse =
    FileHandler.GetTextureByName(textures.DiffuseColor);
Texture2D specular =
    FileHandler.GetTextureByName(textures.SpecularColor);
Color ambientColor =
    FileHandler.GetColorFromString(material.color);

if (diffuse != null)
  material.SetTexture("_MainTex", diffuse);

if (specular != null)
  material.SetTexture("_MainTex", specular);

```

```
26 if (bump != null)
27   material.SetTexture("_BumpMap", bump);
28
29 if (ambientColor != null)
30   material.SetColor("_Color", ambientColor);
```

With the first line in the code above, all materials connected to the FBX file can be found. These materials are then extracted and stored under a specific path in the project's "Assets" folder. When instantiating materials, the *Standard Shader* is applied by default. This shader is physically-based and contains properties mentioned in 4.2.2. Even though the **AssetImporter** is capable of reading the amount of materials connected to the mesh, it can't read any additional information. The **MaterialData** object was created with the second code snippet listed in this chapter. This can now be used to instantiate texture objects and other properties (line 15 to 19), that can now be applied to the Unity material (line 23 to 32). This material can then be applied to the object. The last step of the program creates an AssetBundle from the object and sends it to the Backend system.

4.5 Unity Client and run-time import

Now that the upload process and the FBX-to-AssetBundle conversion is complete, the Unity client can be created. After successfully storing the AssetBundle into the Database, the Unity client is notified and starts a Coroutine to receive AssetBundles.

```
1  IEnumerator GetUnityWebRequest(int objectId)
2  {
3     UnityWebRequest www = UnityWebRequest.Get("IP" + objectId);
4     yield return www.SendWebRequest();
5
6     if (www.isNetworkError || www.isHttpError){...}
7     else
8     {
9        response = www.downloadHandler.text;
10
```

```
11        // Decoding bytes and write to Assetbundle
12        byte[] decodedBytes;
13        AssetBundle bundle = null;
14        try
15        {
16           decodedBytes = Convert.FromBase64String(response);
17           Stream stream = new MemoryStream(response);
18           bundle = AssetBundle.LoadFromStream(stream);
19        }
20        catch (FormatException e) {...}
21
22        if (bundle != null)
23        {
24           GameObject[] assets =
                bundle.LoadAllAssets<GameObject>();
25           foreach (var asset in assets)
26           {
27              // Instantiate object
28              GameObject obj = Instantiate(asset);
29           }
30        }
31     }
32  }
```

The client receives the data as a Base64 encoded string. It has to be decoded (line number 16) before an AssetBundle can be instantiated from a MemoryStream (line number 17).

Two pictures can be seen in figure 17. A grey sphere without any textures is displayed on the left. This appearance occurs when loading a FBX file directly into Unity without conversion. The right picture shows how the sphere is visualized after the conversion. Both the *Albedo* and *Normal Map* property were applied to the material by executing the workflow presented in this thesis.

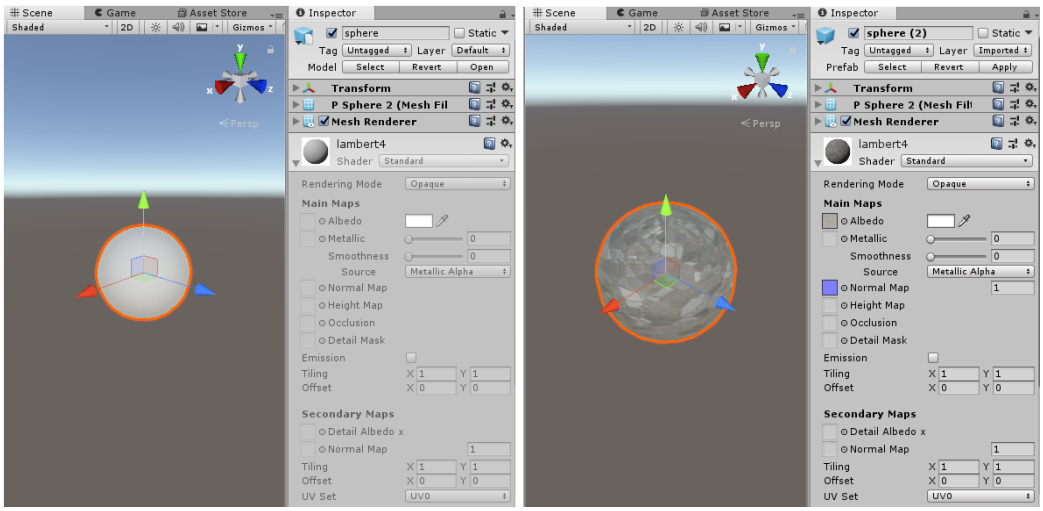

Figure 17: Sphere imported into Unity before (left) and after (right) the conversion.

5 Conclusion & Future Work

Modern shading techniques are very sophisticated and are able to include detailed, real-life physical phenomena. In the process of creating three-dimensional models, artists are usually bound to a specific modeling software, that allows them to create their models any way they like. Models are therefore created with platform-dependent materials and shaders. Importing them into running Unity applications, without preprocessing steps, can be a difficult task. This thesis presented a method to automate the process of setting textures and other properties in Unity materials. The thesis proposes a method for importing FBX files into Unity applications during run-time while retaining material properties. This specific workflow for importing models is only relevant for Unity applications, which want to include FBX models during run-time. The part of reading FBX information with the Autodesk API could potentially be used for other use-cases as well.

With the help of the contributions of this thesis, that are mentioned in the introduction, and depending on the user needs, the scope of this workflow can be extended for future work. A user interface can be created to give the user the opportunity, to select specific parameters for the conversion. Parameters could include the type of shader used by Unity's material. Furthermore, it would be useful to specify, which parameters from the FBX file should be taken over by which parameters on the Unity material. FBX files not only contain material information but also include properties like animations, that have to be applied manually. Even though Autodesk and Unity have a ongoing technical collaboration, import and export processes can still be improved. In order to produce a fully functional interface, that imports models automatically, all FBX components have to be taken into account.

List of Figures

Literaturverzeichnis

[1] Tomas Akenine-Moeller et al. *Real-Time Rendering*. CRC Press, 2018.

[2] Allegorithmic. *Substance Painter (version 2018.1.0)*. 2006. url: https://www.allegorithmic.com/products/substance-painter.

[3] Autodesk. *FBX SDK Programmer's Guide*. Online; accessed 28 April 2018. 2014. url: http://docs.autodesk.com/FBX/2014/ENU/FBX-SDK-Documentation/.

[4] Autodesk. *Maya (version 2017)*. 2006. url: https://www.autodesk.com/products/maya/.

[5] Mike Bailey and Steve Cunningham. *Graphics Shaders - Theory and Practice*. 2nd ed. CRC Press, 2012.

[6] Claudia Doppioslash. *Physically Based Shader Development for Unity 2017*. Liverpool: Aspress, 2018.

[7] Dave Shreiner Edward Angel. *Interactive Computer Graphics*. Boston: Addison-Wesley, 2012.

[8] Adam Lake. *Game Programming Gems 8*. Course Technology PTR, 2010.

[9] Microsoft. *C++ REST SDK (Codename "Casablanca")*. Online; accessed 15 Mai 2018. url: https://msdn.microsoft.com/en-us/library/jj969455.aspx.

[10] Naty Hoffman. *Physics and Math of Shading*. Online; accessed 13 September 2018. 2015. url: https://blog.selfshadow.com/publications/s2015-shading-course/hoffman/s2015_pbs_physics_math_slides.pdf.

[11] OpenStax. *College Physics*. https://legacy.cnx.org/content/col11406/1.9. Online; accessed 20 April 2018. 2015.

[12] Adam Watkins. *Creating Games with Unity and Maya: How to Develop Fun and Marketable 3D Games*. Focal Press, 2011.

6 Appendix

Server and Client class created with Microsoft's C++ Rest SDK *Casablanca*.

Server:

```
int main(int argc, char* argv[])
{
  InterruptHandler::hookSIGINT();

  Server server;
  server.setEndpoint(L"http", 41004, L"/api/v1");

  try {
    // wait for server initialization...
    server.accept().wait();
    std::wcout << L"Modern C++ Server now listening for requests
        at: " << server.endpoint() << '\n';

    InterruptHandler::waitForUserInterrupt();

    server.shutdown().wait();
  }
  catch (std::exception & e) {
    std::cerr << e.what() << '\n';
  }

  system("pause");
}
```

Client:

```
class Client
{
public:
  Client();
  ~Client();
  pplx::task<json::value> getRequest();
  pplx::task<void> postRequest(std::vector<BYTE> file,
```

```
        std::wstring description, std::wstring name, std::wstring
        thumbnail, bool stationary);
};
```