Defining Design Patterns for Pattern Based Material Layering in Real-Time Engines

Matthias Patscheider

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Digital Arts

in Hagenberg

im November 2018

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, November 27, 2018

Matthias Patscheider

Contents

Abstract

Current resources with regard to material layering (pattern layering) do not provide information on how different layering methods affect all levels of a video game production. They focus either on rendering algorithms and implementations or on how to solve particular project specific problems.

In this work a catalog of pattern layering related design patterns is developed which describes solutions to recurring problems in a standardized form. The individual elements form a structured language of interlinked patterns including all important information of a problem within. They present how, when and why to use a certain pattern and what the consequences to be expected are. The goal is to provide better information before these decisions are made: which pattern layering method should be used and how to include it into this particular project.

Solutions proposed in this catalog have been used in different huge video game productions. Most of them have also been evaluated and tested in the VR experience *Letzte Worte* (2019). In conclusion, this catalog can be used to improve information based decision making with regard to pattern layering as well as input and inspiration to create more user friendly tools.

Kurzfassung

Um Entscheidungsprozesse in Hinblick auf Pattern Layering zu vereinfachen, wird in dieser Arbeit ein Katalog an Design Patterns vorgestellt. Diese Patterns bilden die Elemente einer vernetzten und abstrakten Spache zur Beschreibung und Lösung wiederkehrender Proleme.

Bisherige Arbeiten konzentrieren sich primär auf die technische Implementierung oder auf problem- und projektspezifische Lösungsansätze. Diese berücksichtigen dabei kaum das große Ganze und welche Auswirkungen eine Entscheidung auf andere Systeme haben kann. Diese Arbeit stellt ein Werkzeug zur Verfügung, um informierte Entscheidungen zu ermöglichen, bevor man mit den Problemen der einzelnen Ansätze konfrontiert wird.

Die hier dargestellten Lösungen stammen aus verschiedenen großen Videospielproduktionen der letzten Jahre und wurden großteils auch im Masterprojekt *Letzte Worte* (2019) eingesetzt und evaluiert. Dieser Katalog ist ein praktisches Hilfsmittel für den Leser, um bessere Entscheidungen im Bezug auf Pattern Layering zu treffen.

Chapter 1

Introduction

Any complex surface can be recreated by blending or layering an arbitrary number of different less complex base materials.[1] Material layering is becoming an increasingly popular approach in recreating real world surface complexity for video games. Many *AAA* games use this approach for their shading workflow (e.g., [34, 37, 41, 43, 44]).

The growing importance of material layering is accompanied by three big trends within the video game industry. Content creation seems to shift more and more into the game engines. *The Coalition*, *Ready at Dawn* and *Epic Games* transfer a big part of the material pipeline into the engine [30, 17, 19]. *Unreal Engine 4 (UE4)* and *Unity* provide many built in art tools (e.g., the shader graphs). Different 3rd party tools use live links to visualize changes directly within the game engine (e.g., *Motion Builder* and *Substance Painter*). Tools like *Houdini* and *Substance Designer* use plug-ins to enable working from within the engine. Further, a physical plausible workflow is preferred over a visual one. This improves consistency of assets across different lighting setups. It also increases consistency across different artists. Nowadays most 3D applications support a physically based rendering (PBR) shading workflow. Asset creation shifts more and more towards procedural techniques [32].

There are many different approaches to layer and blend materials. Each one has its own advantages and disadvantages. The different methods will be explained later in this work. A major investigation focus for the analysis of the material layering methods for interactive, virtual environments[2] is about balancing visual appearance, technical constraints and limited production resources. Decomposing a complex surface into smaller, manageable elements is good way to split a task into smaller, less complex parts. Figure 1.1 shows an example of how a complex material can be recreated by combining a set of more generic materials. Splitting up a surface into smaller components can be beneficial from an artistic, production and technical point of view. Subproblems can be handled individually.

With this work, I will provide a proposal for material layering related design patterns, to support the decision making process while creating layered materials. To further

[1] The term *material* can have a variety of different meanings within computer graphics and beyond, see section 2.3.

[2] In this work an interactive, virtual environment is an artificial world provided by a computer. Its goal is to provide for an completely immersive world that is different from the players physical location [14, p. 9, 30][11, p. 743].

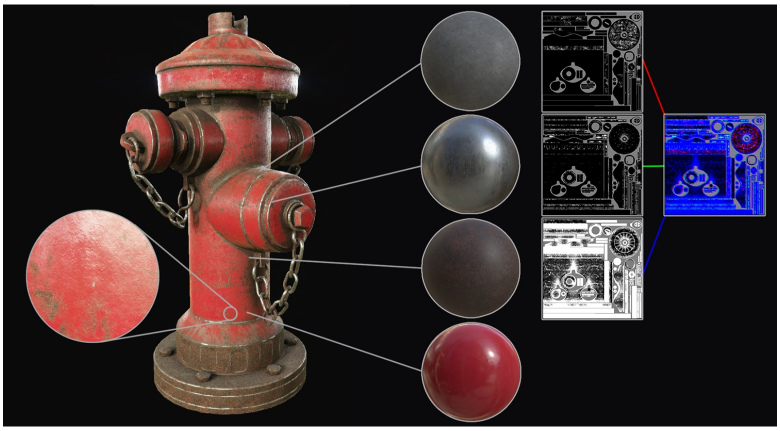


Figure 1.1: Shading with pattern layering. This image illustrates the basic concept of the pattern layering. Pattern layering represents the most relevant material layering method for real-time applications. The fire hydrant is composed of 4 different tillable base materials. These base materials (dirt, clean metal) are blended by using the masks on the right. Image source: [31].

elaborate why material layering is an important technique for creating large scale virtual environments, I will give a brief example. Looking at the ruins of an old castle, you might see that there is a lot of complexity going on over a huge amount of space. There are probably different layers of stones, plaster, color and moss and all these layers blend together and show the history of the object. A real world example can be seen in figure 1.2.

The wall surface[3] indicates the age, usage, interaction: Was it exposed to nature? Was it taken care of or simply left to decay? Capturing this kind of object history is more than just creating an interesting visual. It plays a major role in creating a deep, interesting and appealing world. Decomposing the material into different independent base materials and analyze how each material is blended with the others makes the challenge much more manageable than trying to solve every problem at once. A side effect of creating solutions to smaller, less specific problems is the possibility to reuse and modify the components and share them. This is normally not possible for all-in-one solutions as every subpart is heavily interlinked.

The visual appearance is only one part, a completely different thing is to make the game run smoothly in real-time. The available resources—e.g., memory, bandwidth, CPU, GPU etc.—are limited and have to be used efficiently to make this possible. Depending on projects, requirements and hardware, different components will create a bottleneck in the rendering pipeline. Staying with the example of a big castle wall, the first approach might be to texture the huge surface by covering it entirely with huge unique textures. This approach relies on giant texture files and is therefore risky to create a bottleneck either by extending the available memory or bandwidth. Another approach is to recreate the micro detail of the wall with actual 3D geometry. This method might work for a few objects but soon result in an enormous polycount and

[3]The term surface is further specified in section 2.1.

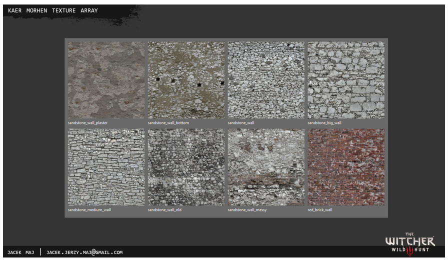
Figure 1.2: A challenging surface to recreate virtually. Recreating walls like this from the *Heidelberg Castle* are a challenging task for 3D interactive environments. The main challenge is the rich amount of detail and surface variation across such a huge area of space. Recreating this richness of the surface with the stones in varying sizes, plaster, color and dirt is challenging from an artistic as well as a technical point of view. This becomes even more challenging as the player can move freely and seamlessly around, seeing the castle walls from near and far. One solution is to split the surface into different more generic base materials.

create a bottleneck in another part of the rendering pipeline. A probably better suited solution might be to use material layering. One way to do material layering is to use several tillable textures, blend and manipulate them with different methods and create a complex, large scale surface. *The Witcher 3: Wild Hunt* [34] used a similar approach to create wall materials as shown in figure 1.3.

1.1 Research Objective

The goal of this thesis is to collect answers for recurring decisions regarding material layering in real-time applications. This thesis will cover bigger concepts behind common texture layering problems and is not intended as a step-by-step guide in how to implement a certain design. It will provide users with a tool set of tested methods to improve decision making concerning pattern based material layering. The different approaches to material layering are discussed in section 3. In the end, the design patters should support game makers in populating a scene with a huge amount of appealing, distinctive and complex objects without extending the technical limitations of a first person real-time application. To do so, this work attempts to define design patterns to answer recurring problems regarding pattern based material layering. As a template for the pattern design I adopt the scheme presented by Erich Gamma et al. [10] and Jenifer Tidwell [21]. According to this template, design patterns themselves should contain the most important information like application, benefit and risk [10, p. 3]. See chapter 6 for further details on design patterns. *Weta Digial* evaluates new technologies and tools

(a)



(b)

(c)

Figure 1.3: Multitexture materials used for the buildings in *Witcher 3* [34]. Figure (a) shows the castle *Kaer Morhen*. The walls of the castle are textured by layering different smaller scale textures, shown in figure (b). In combination with different masks, layering and parametric methods, it is possible to ensure a high texel density even from a closer view (c). Image source: [29].

according to three major aspect: “performance vs. correctness vs. artistic freedom” [22]. These same free factors play an essential role for evaluating the design patterns defined in this work;

Artistic Freedom: To what extend does the technique enable artists to reach a visual goal? What are the aesthetic limitations and downsides of the used method that are to be expected?

Performance: What is the impact of the method on performance? Which bottlenecks might the method lead to (e.g., memory, bandwidth, CPU, GPU). Might this method solve a performance issue I already have?

Correctness: How versatile, flexible and robust is the used method? Does it support an interactive, shared workflow and can sub-parts be reused, combined and shared?

All test cases are based on opaque, physically based,[4] dielectric and metallic materials,

[4]The term physically based is used to describe methods that try to imitate the real world behavior by following the laws of physics [20, p. 1]. In general, you can distinguish between physically correct and

not including advanced shading concepts like translucency, transparency, subsurface scattering and volumetrics. To make the results comparable, all test share a similar aesthetic goal in creating a photorealistic[5] and vivid environment. Most design patterns were tested and evaluated in the production of *Letzte Worte* [42]. Neither the creation process of textures, their blending in a texturing software with the result of a single, baked texture nor the technical implementation are going to be part of the thesis.

1.2 Structure of this Thesis

Chapter 2 starts by defining the key terminology for this work. Chapter 3 provides an overview of the current development and state of the art. It gives an overview of the different concepts for material layering. Further, it discusses which methods are already relevant for real-time applications and which might become interesting in the near future. The thesis continues by defining an abstract description model for pattern layering in chapter 4. This will be especially relevant to understand the structure and categorization of the design patterns. Chapter 5 focuses on the implementation of the previously theoretically described models. Before digging into the core of this work, a short spin is made to discuss design patterns from other industries. Chapter 6 explains how these concepts can be transposed to this work. Chapter 7 presents a detailed catalog of design patterns. They are structured hierarchically and categorized thematically. This helps to easily navigate to the problem relevant patterns. Finally, chapter 8 concludes by presenting the results, limitations and prospects for future research.

physically plausible approaches. While the former tries to stay as close to reality as possible, the latter deliberately chooses art-directability and performance before physical accuracy [5, p. 12][61, p. 2].

[5] The appearance of a photo realistic object should be close or indistinguishable from photographs. As human perception is subjective, this may vary from person to person [20, p. 4][1, p. 99].

Chapter 2

Real-Time Rendering Definitions

The terms such as for example surface, texture and material have various meanings depending on the context they are used in. Definitions from other scientific fields like engineering, chemistry and physics will not necessarily cover the same needs as in computer graphics. Defining a material as "[. . .] a substance that things can be made from." [62] does not make a lot of sense for the virtual concept of material in computer graphics. Another issue is that even in computer graphics and across different 3D applications the terminology is often used inconsistently. Even throughout the same applications (e.g., *UE4*) I found the use of the term material non-specific and referring to many different concepts. It is therefore essential for the clarity of this work to define the terminology used in this work first.

2.1 Surface and Surface Area

In the present work *surface* is used to describe the visual appearance of an object and the way it interacts with lights rather than the most outer area of an object in a geometrical sense. It refers to the different visual properties of an object like the age, history and usage of the castle wall (see chapter 1). To refer to its secondary meaning of the "the outside part or uppermost layer of something" [62], I will use explicitly the term surface area to make a clear distinction between both terms.

2.2 Texture

A texture is often referred to as a color image that covers the surface area of an object, while, in practice, textures can contain all kinds of information influencing the shading equation. This work relies on the definition for texture defined by Thomas Akeninne-Möller et al. [1, p. 32–36]: "[A texture] can be thought of as any large array of data." This statement already includes many properties that textures can influence. This array of data can contain the color data that is to be projected onto the mesh, but it can also contain all kinds of other shading information. Besides they are generally used to define a vast variety of surface properties by affecting the shading equation [1, p. 180–181] of a material. Most of all bigger surface shading description is stored in this way. Joe Wilson [69] gives examples of surface properties that are most often—but not exclusively—

(a) (b) (c)

Figure 2.1: Using textures to manipulate UV coordinates. These images shows a less obvious example for using textures. The texture is not used to define a surface property but to manipulate the UV coordinates. Figure (a) shows a representation of the UV coordinates. The red and green value represent the U and V coordinates reaching from zero to one. Figure (b) shows a perlin noise texture that is used to manipulate the UV Coordinates within the shader. Figure (c) shows the manipulated UV coordinates that distorts the texture projection (e.g., base color, roughness) and can even be used to create animated shader effects.

defined by textures:

Diffuse: These textures define the surface color and can be slightly different in name and stored information data depending on the application and shading model they are used for. The most commonly used names for diffuse textures in physically based shading are base color, diffuse and albedo. This work uses the term base color.

Microsurface structure: The two most commonly used maps to influence this property are roughness and glossiness maps. They basically define how rough or smooth a surface appears by influencing the shape of the specular reflection.

The amount of reflected light: The most important texture inputs to influence this material property are metalness, specular and index of reflection [69].

As mentioned before, textures are not always projected onto an object but can also be used as an object independent "stand-alone data table" [11, p. 462] as pointed out by Jason Gregory. Matt Pharr et al. [20, p. 597] choose an even broader definition for a texture:

> [...] a texture is a fairly general concept: it is a function that maps points in some domain (e.g., a surface's (u, v) parametric space or (x, y, z) object space) to values in some other domain (e.g., spectra or the real numbers).

2.3 Material and Shading

The terms material and shader are closely connected with one another. The use of both terms seems inconsistent across different applications such as for example *Unity* and

UE4. It is therefore especially important to define what those terms mean in the course of this work. Materials are defined by Akenine Möller et al. [1, p. 468] as follows:

> A material is a complete description of the visual properties of a mesh. This includes a specification of the textures that are mapped to its surface and also various higher-level properties, such as which shader programs to use when rendering the mesh, the input parameters to those shaders and other parameters that control the functionality of the graphics acceleration hardware itself.

The term shader does include all these shader programs: the input manipulation within the shader code and shading graphs. This definition also allows to give statements regardless of the underlying shading language and hardware. The terminology for texture, material and shader within *Unity* applies to this work. A shader within *Unity* is defined as follows [65]:

> A Material specifies one specific Shader to use, and the Shader used determines which options are available in the Material.

This simplified definition of a shader will work for most parts within this work. Materials within *UE4* are defined as follows in contrast the prior definition [56]:

> Materials are defined by a set of states that control how the material is rendered (blend mode, two sided, etc.) and a set of material inputs that control how the material interacts with the various rendering passes (Base Color, Roughness, Normal, etc.).

Materials within *UE4* can perform both functionalities of a shader as well as the model specific descriptions—see definition above. An *UE4* material can contain certain descriptions of how to evaluate the shader equation, but it can also be applied to a mesh directly. To avoid further confusion, this work will use a strict separation between *UE4* material instances and materials. *UE4* materials are strictly used as shaders. The material instances, on the other hand, are used as materials and contain the object specific values and textures. This aligns with the distinction between shader and material found in *Unity*. In terms of this work, *UE4* materials will be strictly used and referred to as shaders while material instances as materials. Other meanings for materials within this work are:

Base Material: Base material refers to a distinctive part of a surface area that differs in its physical properties. In contrast to material container, it describes the visual and artistic categorization. A surface can be split into different generic base materials. These base materials can be used to recreate a similar surface to the initial reference, for example. The granularity of splitting a surface into different base materials is defined by the artist and the final purpose. In a complex environment a base material may refer to cobblestone as well as the more detailed individual layers such as stone, mud, pebbles etc.

Material Container: A material container is a technical component of the material layering model (see chapter 4). In contrast to base material, it may not only represent the entire surface properties but may only contain technical parameters and variables.

2.4 Summary

The purpose of this chapter was to define the fundamental terminology for this work. This is necessary as terms like texture, material and shader are used inconsistently across different applications and scientific fields. For the purpose of this work, a texture is defined as a stored set of data that can be mapped onto any object. It is used to influence the shader equation within the rendering process. A texture is therefore used to define certain surface properties like color, light interaction, blending mask or texture mapping independent lookup tables.[1] A shader defines which parameters are available within a material. Every material has to specify one shader, while shaders can be assigned to an infinite amount of materials. The shader defines which parameters and textures are available within the material and how the material is processed and rendered on the GPU. A Shader defines the available inputs and the different shader programs used. Materials are directly applied to objects within the 3D scene and define the object specific inputs.

[1] Lookup tables represent arrays of data. They are used in computer graphics to avoid heavy computation and use stored data with array indexing operations instead [68].

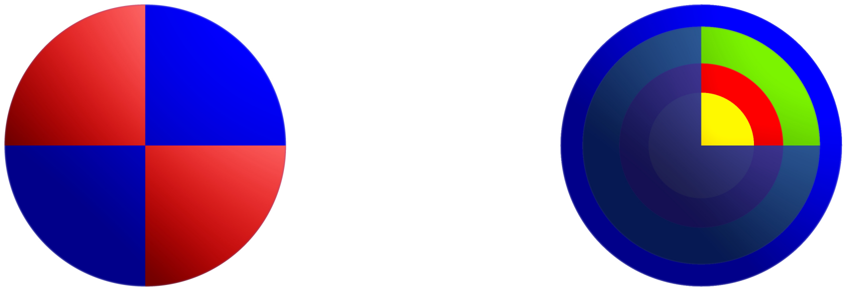
Chapter 3

Material Layering

This chapter is about the current state of material layering methods. The first part provides a definition of material layering and a categorization for these methods. As the the modern game rendering pipeline is a complex system of many interlinked stages, I will try to see material layering in a larger context and not as an independent isolated component. The methods presented in this work are applied in various stages of the creation and rendering pipeline. This includes the asset creation process (e.g., modeling, texturing, normals) as well as shading and rendering (e.g., vertex shader, pixel shader, post processing).

(a) (b) (c)

Figure 3.1: Comparison of three different shading methods. Figure (a) and (b) show different implementations of material layering: *BxDF layering* and *pattern layering*. Image (c) shows the reference of using traditional baked texture maps. The shading was done in *Blender*, model and textures are from the *Substance Share* website [46].



(a) (b)

Figure 3.2: Comparison of material blending (a) and material coating (b). In material blending, masks are used to define where a base material is applied. They do not represent the physical properties of layers stacked onto one another but rather a blend defining the influence on the final output. Material coating (b) simulates an actual coating or stacking of different layers onto one another. The purpose is to recreate the complex scattering of light when passing through different layers. This is not taken into account in traditional BxDF or rasterization shading models.

Similar visual results can be achieved with different methods. An example can be found in figure 3.1. For this example, I recreated the material setup for *Allegorithmics* fire hydrant using three different methods. The images show the different material layering methods: pattern layering and BxDF layering. Both will be explained later in this chapter. The last image shows the reference image using pre-baked maps, i.e., there is a single texture for the individual material parameter inputs already containing all the information of the different base materials. The mesh and base textures used can be downloaded from the *Substance Share* website [46].

3.1 Definition

This section provides a brief definition of what material layering means in the course of this thesis as well as the various methods it can be referred to in different applications and contexts. Material layering is the process of either blending distinctive base materials or recreating the physical layers of real world surfaces, illustrated in figure 3.2. The former is used to create variety in different surface types; the latter is used to recreate the complex lighting effect within thin physical layers. It simulates the light scattering on, between and within physical material layers. The approaches are not directly linked to one another. Both ideas have different use cases, tools, algorithms and shaders. A distinction between the two approaches is therefore essential.

Material Blending: It describes the process of blending two distinctive base materials. This process does not take the complex process of scattering within the layers or at the layer boundaries into account.

Material Coating: This approach simulates the complex effects going on when stacking a—most probably translucent—layer on top of another layer. This means that the output of the first layer affects the input of the next layer.

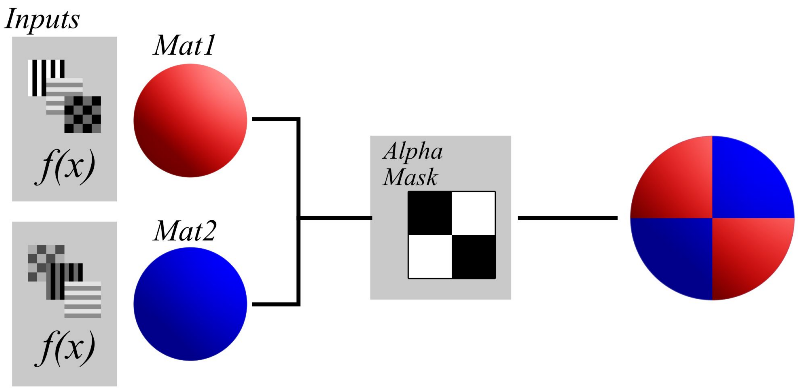


Figure 3.3: *Color layering* with two base materials. The process would look the same if applied to more base materials. Either material may have independent textures, inputs and shaders. The individual materials are computed independently from one another. Finally, the rendered output layers are blended together by using the alpha channel as a mask. If additional data like displacement is used, the blending process is more sophisticated than simple alpha blending (e.g., blending displacements properly [63]).

3.2 Categorization

Davide Pesare—a senior software engineer—gives an overview of the commonly used material layering methods on his webblog [63]. He distinguishes between four different types of material layering: *color layering*, *pattern layering*, *BxDF*[1] *layering* and *illumination lobe based layering*. He provides a useful guide to better understand what different approaches for layering materials exist and how they differ.

The most relevant one for this work is pattern layering as it is technically already being used for real-time applications. BxDF layering is an approach that is highly connected to ray tracing and therefore not yet applicable for the purpose or real-time applications. Nevertheless, they might become suitable for this in the future as hardware is evolving and development in real-time ray tracing is already happening as shown by a collaborative demo of *Epic Games*, *NVIDIA* and *ILMxLAB* [51]. A paper by Belcour [4] has just recently been released showing an approach that makes illumination lobe based layering suitable for real-time applications. More publications and implementations in this direction might therefore follow in the next years.

3.2.1 Color Layering

Color layering was extensively used before physically based rendering (PBR) was introduced. A set of different materials gets applied to the same object. An independent *RGBA* layer is calculated for every material. These render layers contain the final object color with all illumination baked in. Finally, the different render layers for each mate-

[1]The x in *BxDF* is a wildcard character and can stand for all kind of different *BxDF* functions; BSDF (bidirectional scattering distribution function), BSSRDF (bidirectional scattering-surface reflectance distribution function), BRDF (bidirectional reflectance distribution function) and BTDF (bidirectional transmittance distribution function) [67].

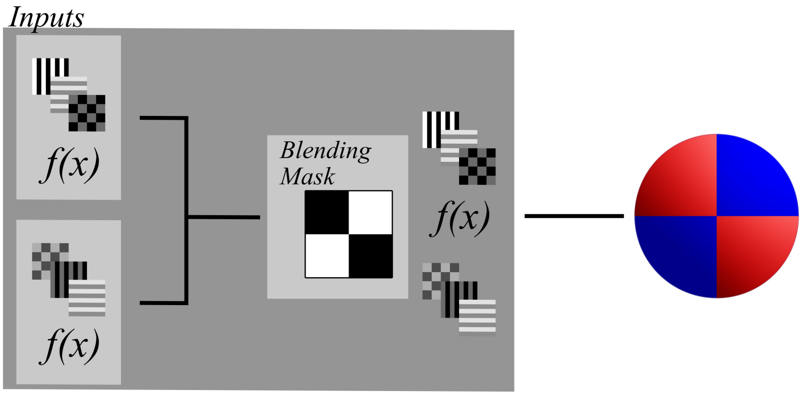


Figure 3.4: *Pattern layering* with two base materials. Both base material are described by independent inputs. In contrast to *color layering*, the parameter inputs are blended instead of the shader outputs. A more detailed overview of how pattern layering works can be found in chapter 4.

rial are blended together based on their alpha channel. Figure 3.3 shows the schematic process of this material layering method. It can also be expanded to include advanced shading information like displacement layering [63]. Color layering results in an object being redrawn for every base material.

3.2.2 Pattern Layering

The idea behind pattern layering, in contrast to color layering, is to blend the material inputs within the shader instead of their outputs. The inputs (e.g., base color, roughness, metallic) get blended with one another. Finally, the blended parameters define how the material effects the corresponding rendering pass (rasterization renderer) or illumination lobe (for a ray trace renderer). This method assumes that all base materials use the same illumination model. The inputs of the final shading programs are identical for the materials. They can therefore be packed together and send to the GPU as single job. This method reduces the material count drastically in comparison to color layering and is therefore more efficient. The features for the individual materials are limited by the shader; only inputs that are defined by the shader can be used for material layering [63]. To circumvent this issue, *Epic* and *The Coalition* have worked on systems that generate a shader dynamically depending on the given inputs (see section 5.2.3). More detail on how *UE4* and *Unity* handle pattern layering and its constraints is provided in section 5.2.

Pattern layering is a flexible method. All kind of data types (e.g., textures, procedural noises, meshes, scene data) can be used to blend and manipulate the individual base materials. For recreating a surface that is partially covered by water, many approaches could be applied. Blending all texture maps for each and every render pass is one possibility. Another approach is to use a procedural noise that influences the roughness, base color and normal parameter inputs and creates a similar result. Pattern layering does therefore not only refer to the combining of several base materials into a layered material but also to manipulating input parameters to create a similar effect. The blending of the different parameters can be complex and therefore easily produce unintended values. Ensuring a proper blending of all inputs is the biggest challenge for a pattern based material layering system. Common issues arising form pattern layering are: a wrong roughness and index of refraction (ior) accumulation, multiple values that influence a single output and issues with blending normal vectors. These issues are further discussed in section 4.4.1 on page 26.

3.2.3 BxDF Layering

Bxdf layering is highly related to ray tracing. The layering system provides the renderer with the information related to which material covers which parts of the object. The blending affects the *probability density functions* of the ray and is therefore independent from the individual materials themselves. The *probability density function* defines the probability with which an incoming ray hits a certain material. A black mask defines the probability of 0% for a ray to hit the assigned material while a white mask is equal to a probability of a 100%. Grey value represent a probability between a 0% and a 100%. As shown in figure 3.5, the probability for the black masked area on the left side is a 100% to hit the blue *MAT1*. Going further and further to the right side of the object, this probability shifts more towards the red material *MAT2*. The probability increases until it is finally a 100% at the white masked areas.

According to Davide Pesare, the additional expanse for this approach will be minimal compared to pattern layering when the light prediction algorithms for path tracer improve [63]. In the fire hydrant example (see figure 3.1), when comparing both methods, the rendering time for BxDF layering increased dramatically compared to pattern layering. The huge advantages of BxDF layering are: the individual materials are not limited by the constraints of a single shader, the blending is physically more accurate than pattern layering and the need for complex pipeline tools to assure the proper blending of individual lobe inputs disappears. By only influencing the probability density function the materials are completely independent from one another. This removes the technical issues for pattern layering—mentioned before—when blending different materials. Although BxDF layering is more versatile and less complex than pattern layering, it is still not able to recreate complex stacking of different materials, as explained by Andrea Weidlich et al. [22, p. 9]:

> Materials (i.e., BRDFs) can be stacked, but only with an opacity channel; they do not influence each other. This means that a rough surface on top of a smooth surface will not increase the roughness of the underlying surface (although this would happen in reality).

3.2.4 Illumination Lobe Based Layering

Before going into detail about illumination lobe based layering, I want to talk briefly about lobes in the context of shading. A BxDF is composed of different lobes that define how the material interacts with light. Figure 3.7 illustrates this fact. *Pixars PxrSurface* is an *Uber Shader* dedicated to replicate all kinds of different materials. To

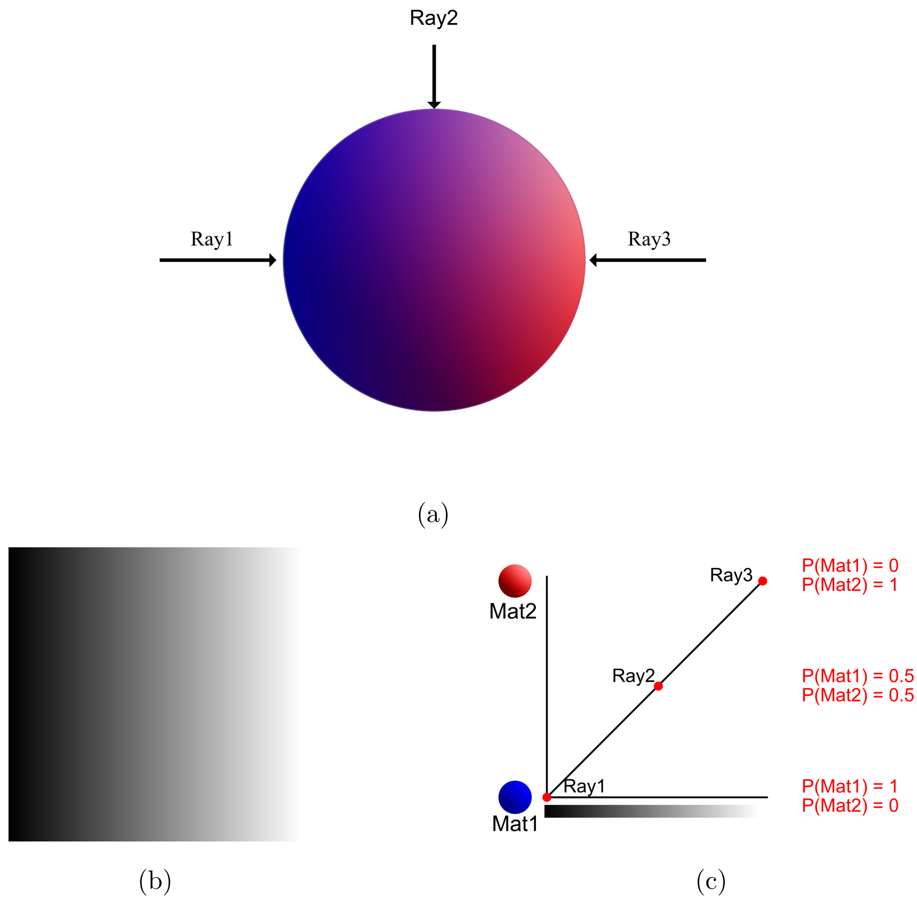


Figure 3.5: *BxDF layering* with two base materials. Figure (a) shows a shaded sphere with two BxDF materials blended together. They are combined using a linear gradient from black to white as seen in figure (b), from the blue material *Mat1* to the red material *Mat2*. Rather than blending the output of both materials or the inputs, this method influences the probability density function of the render equation for the incoming rays. As shown in figure (c), the probability for the black masked area on the right side is a 100% to hit the blue *MAT1*. Going further to the right side of the object, this probability shifts more towards the red material *MAT2*. The probability increases until it is finally a 100% at the white masked areas.

make this possible, the material properties are described by ten different lobes (e.g., diffuse, three specular, glass, subsurface etc.). For the basic BxDF shading these lobes are mixed linearly together. The principle behind illumination lobe based layering is to stack different base materials that use a different set of illumination lobes. These lobes are than stacked onto one another by also replicating the physical thickness of a material layer.

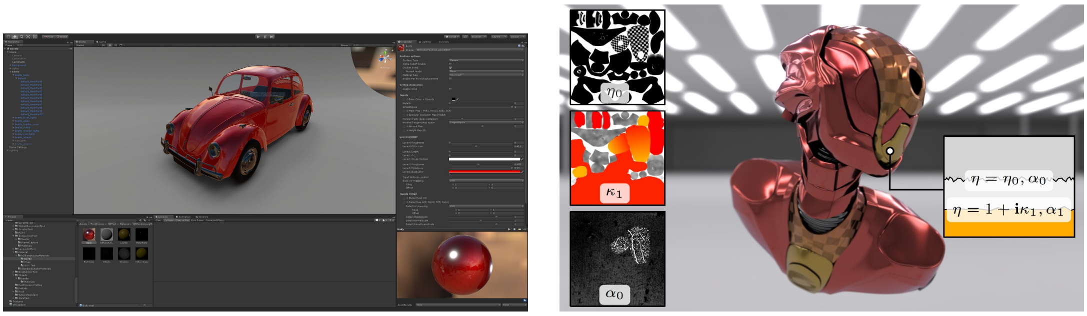
All the previously explained methods are able to yield astonishing results but fail to simulate the light propagation in a physically plausible way. The illumination lobe base layering is the only approach presented in this work that aims to imitate the real world behavior of different, distinctive layers stacked on one another. Wenzel Jakob provides some examples for surface properties that are not possible to recreate with the traditional BxDF shading models. Some of these examples are: ceramic covered by glaze, colored car paint with an additional layer of clear coat and biological layered materials like skin or leafs [12, p. 1]. He further explains the challenging task of recreating this complex real world materials in following paragraph from [12, p. 1]:

> Simulating these types of layered materials in renderings is surprisingly difficult: when a quantum of light enters the top layer, it can undergo a complex sequence of scattering events within individual layers and at layer boundaries; finally, the light is either absorbed or able to leave the material. The details of this intermediate scattering process are important, since they determine both the intensity and distribution of scattered light. [. . .] Even the simplest nontrivial system, of a single medium bounded by a smooth interface, is only roughly approximated by standard BRDF models [. . .].

Recreating this complex effect has been a subject of interest over the last years. A lot of solutions have been proposed and adopted to recreate a really specific case. *Tencent* and *Epic Games* have shown an astonishing result for a real-time skin shader using an additional glossy specular layers stacked over the base shader and an additional specular lobe underneath it in [64]. Estevez et al. [7, p. 1] propose an efficient method to simulate the back scattering characteristics of fabrics by introducing an additional sheen specular lobe. The current methods used for real-time productions are limited to one specific kind of material, e.g., skin, fabric or coated paint. Therefore, a lot of research has been going on to create an illumination lobe based material layering system that works for arbitrary base materials and any number of layers.

One attempt do do so was developed by Jakob et al. [13]. They developed a framework to compute layered materials for arbitrary isotropic and anisotropic layers with smooth or rough boundaries of conductors and dielectrics. Their paper has been the starting point for a lot of ongoing research. Their layered material system has just recently been expanded by Zeltner et al. [23] to cover reflective, transmissive, anisotropic layers as well. The limitations of these approaches are their heavy reliance on precomputed and stored data. These methods do not support varying texture materials inputs as they rely on precomputed data.

Belcour has recently released a paper [4] containing a real-time compatible new framework. This work targets a comparable visual quality to preview methods proposed by Jakob et al. [13] and Zeltner et al. [23]. In contrast to their work, Belcour's method, presented in [4], does not rely as heavily on precomputed tabular data but relies rather on statistical analysis and decomposes "light transport into a set of atomic operators that act on its directional statistics". Finally, also this approach relies on stored data but with a much smaller footprint. The paper contains an implementation example for both offline and real-time rendering. Both share the same code for the greatest extent. The real-time implementation has some limitations (e.g., restricted to three base materials and two output lobes) regarding the increase of efficiency. Other limitations common to both renderer are: In cases with several glossy materials, the specular lobes of the output gets inaccurate, anisotropic and subsurface scattering is not implemented, only



(a) (b)

Figure 3.6: Illumination Lobe Based Rendering in both a real-time (a) and offline renderer (b). Image source: [4, p. 73:11].

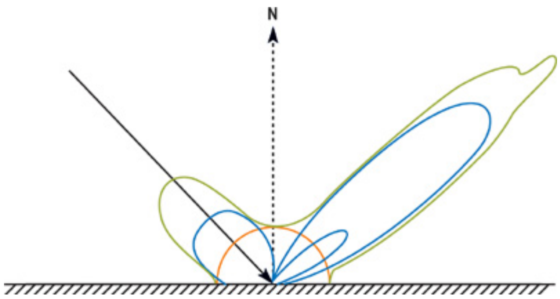


Figure 3.7: A material composed of four different lobes. The orange line represents the diffuse component, the three blue lines are additional lobes (e.g., specular lobes) and the green line shows all the lobes combined that into the BRDF. This is a visible representation of the reflectance at each and every point for every possible incoming and outgoing angle. Image source: [8].

the GGX algorithm for the specular reflection is implemented and the approximation that light enters and exits the material at the same point is made. Belcour specifies which technical criteria a layered material approach has to fit to be suitable for production [4, p. 73]:

> A key difficulty is to provide a realistic model that works with an arbitrary number of layers (possibly textured), accounts for multiple scattering, is energy conserving, requires little storage, has a short precomputation time, supports good importance sampling and is symmetric with respect to light transport evaluation (to be compatible with bidirectionnal rendering techniques).

3.3 Summary

In this chapter, different methods of material layering were discussed. An important differentiation was made between material blending and material coating. In contrast to material blending, material coating is not used to blend different base materials but does rather simulate the complex scattering of light happening between multiple thin layers. A further categorization of material layering methods was made based on an article by Davide Pesare [63]; *color layering*, *pattern layering*, *BxDF layering* and *illumination lobe based layering*. Table 3.1 summarizes the key differences of these methods. Color layering and illumination lobe based layering are not officially implemented in any of the game engines. Color layering can be achieved by using custom render passes. This is expensive as the material needs to be rendered several time. Therefore, it is not used for material layering in real-time applications. Belcour proposes a method to implement a simplified illumination lobe based layering system into a real-time engine in [4].

Table 3.1: Overview of the material layering methods.

Material Layering Method	Material Coating	Game Engine Support	Description
Color layering	-	~	Each material is rendered independently and stored in an RGBA layer. The render layers are blended together by using the alpha channels. There are some advanced techniques to account for complex shading techniques like displacement blending.
Pattern layering	-	X	The idea behind pattern layering is to rather blend the material inputs then their outputs. The different inputs from one base material like base color, roughness and metallic get blended with the inputs of another base material and combined into one material by the shader. Finally, these blended parameters define how the material affects the corresponding rendering passes.
BxDF layering	-	-	BxDF layering does neither blend the inputs nor outputs of the base materials but rather influences the probability density function within a ray tracer. This means it influences the probability with which an incoming ray hits a certain material. The materials and the blending process are independent from one another.
Illumination lobe base layering	X	~	Illumination lobe based layering is the only method that accounts for the complex process of light passing through different layers and being manipulated by them. It imitates the complex scattering and modulation of light when passing through different layers with a thickness and different physical properties.

Chapter 4

The Material Layering Model

The next chapters focus exclusively on pattern based material layering systems and shaders. Therefore, material layering will always refer to pattern layering if not further specified. For the purpose of this work I developed a theoretical and implementation independent schematic model to describe material layering systems and shaders, the *Material Layering Model*. This material layering model is inspired by the high level material layering systems form *UE4* 5.2.3 and *The Coalition* [19]. It splits layered materials in different encapsulated components that are responsible for specific tasks. The main components of this material layering model are: *Material Container*, *Masking Container* and *Blending Module* (see figure 4.1).

Material Container: The material container is an independent component in the material layering model. It represents an encapsulated module that uses different input data and outputs a material layer object.[1] The computations taking place within this component can be complex but do not influence the other components of the material layering model. Like a class in programming, the implementation can change without effecting other components as long as the inputs and outputs stay the same. For most of this work it is enough to interpret a material container as a blackbox that inputs different data, does some computation based on them and exports a material layer object.

Masking Container: The masking container is responsible for generating a 0 to 1 mask that defines the influence intensity and area of the blending module. The masking container can be seen as encapsulated module inputing arbitrary data and optional material layer inputs. The masking container specifies the needed inputs and the logic that is used to derive the 0 to 1 masking information.

Blending Module: The blending module is responsible for the actual blending process of the individual material containers. The blending module inputs the material layers from the material containers and the mask output from the Mask Container. The blending module outputs a new material layer. As mentioned before, the 0 to 1 output of the masking container defines the area and intensity in which the blending takes place. The blending module includes the information of how to blend the individual material layer channels. The material layer output of the

[1]The material layer object is a data type that describes the standardized material properties by defining the individual channel values influencing the corresponding render passes.

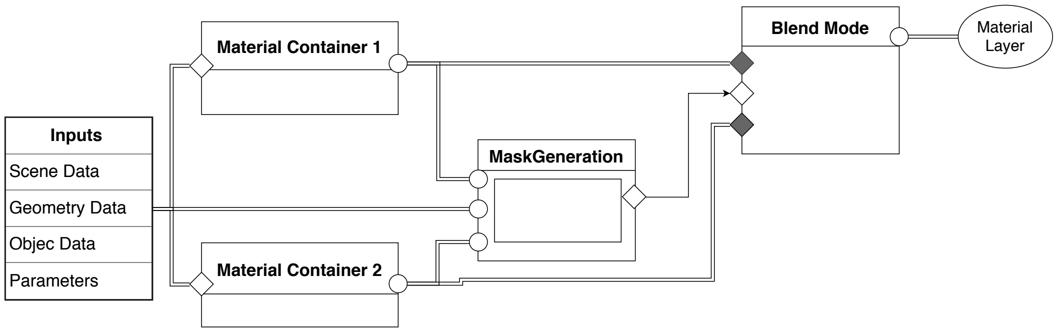


Figure 4.1: Material Layering Model.

blending module can represent the final Shader output, a new material container for further layering or an intermediate result for further manipulation within the shader.

As already mentioned before, this is a theoretical and schematic model to establish a language that can be used for the design patterns in chapter 7. It is intended to work for the different material layering systems within this work. It does neither define how those systems are implemented on a technical level nor how material layering should be implemented. The material layering Model works for either describing a layered material shader or a layering system. In the former case, it represents the structure within the shader that enables the layering of different base materials. In the latter, it represents the structure of the system that is used to compute the final shader from the inputs.

Layered Material Shader: The shader defines the amount of components (e.g., material containers) and the possible inputs. Adding additional inputs and options, changing or replacing individual components or modifying the blending behavior needs to be done manually on the shader level.

Layered Material System: Generally, the layered material system is implemented as a tool—normally within the engine—that uses different encapsulated modules, e.g., *Material Layer* (material containers) and *Material Layer Blends* (combining mask containers and blending module) in *UE4*. These components can be combined easily within the tool. The layered material system does generate the shader code automatically in the background. Individual components can therefore be added, replaced or changed easily without any additional work.

4.1 Inputs

Inputs define which data is accessible from within the different modules of the material layering model. This section is mostly relevant for the material container and masking container components. These components have full access to all input data exposed by the engine: parameters, scene data, object data and mesh data. The material layering model does not define inputs as an independent sub module but as inherent part of the other modules. Which input data can be accessed from the shader is defined by the shading language, engine and tool that is used to create the shader. An overview of all nodes and inputs for *UE4* can be found in its documentation.[2] A huge amount of different data can be accessed, some of them are specific to special use cases. The following sections do not include all possible inputs but, in my opinion, the most useful and universal ones.

4.1.1 Parameters

Parameters are variables that get exposed from within the shader or layering system. They can be set and modulated on a per material or component basis, depending on the implementation. This enables the user to create multiple distinctive materials using the same shader. These parameters can be global or specific to individual components. The available data types for parameters are defined by the shading system. The shader graphs of *UE4* and *Unity* support different data sets. All supported and predefined parameters available in *UE4* shader graph can be found in the documentation.[3] The most important ones are: textures, vectors, scalars and booleans.

Textures are the most common way to store larger data sets for individual materials including color information, information about the lighting interaction, vectors, masking and many more. They are efficient in processing as they represent a simple data table than can be read and does not need heavy computation. Algorithms and workflows, like dynamic texture streaming, channel packing, mid mapping, compression and other optimizations make it possible to load a huge amount of bigger textures into a 3D scene. Nevertheless, the memory in the VRAM is limited and needs to be used carefully. Sending and managing the data and render jobs to the GPU (draw calls) can lead to a bottleneck on the CPU. It is therefore important not to go overboard with the amount of textures and especially the texture resolutions. Vectors, scalars and booleans are incredible versatile and powerful as they can be used to define, drive and influence an infinite number of different operations. Vectors and scalars can be used to define and manipulate colors, directions, normals, switches, remapping, intensities, rotations, vertex positions, uv coordinates etc. Most of the parameters can be changed and assigned at runtime. This works for all parameters that do not enforce the shader to recompile (e.g., static switches in *UE4*).

4.1.2 Scene Data

Different data from the 3D scene can be accessed in addition to the previously mentioned inputs. Some examples are are: mesh data using the corresponding material, data about the object owning the material and different arbitrary scene data exposed by the engine. Some examples for the latter are: data connected to cameras, lights, render passes. The separation between object and mesh related inputs is important as the usages for these data types are different. The mesh data is independent from the 3D scene. So for instance, object related values like hierarchy, position, rotation, scale do not have any influence on them. All of the identical meshes within the 3D scene share this data. All the material inputs are the same for those meshes. Object related inputs, on the other

[2] See the list of all nodes and inputs accessible from within the *UE4* shader graph: https://docs.unrealengine.com/en-us/Engine/Rendering/Materials/ExpressionReference.

[3] See the documentation about parameter expressions: https://docs.unrealengine.com/en-us/Engine/Rendering/Materials/ExpressionReference/Parameters.

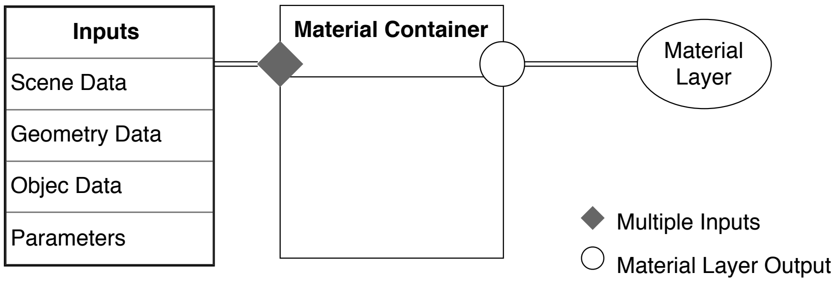


Figure 4.2: Material Container.

hand, are related to the position and hierarchy within the 3D scene. Therefore, inputs can be both influenced or independent from the mesh data.

Mesh Related: These inputs are only influenced by the mesh, e.g., UV coordinates, face and vertex normals, objects space position and object space normal.

Object Related: In contrast to mesh related input, they may differ across objects even by using the identical mesh, e.g., world space normals, world space position, object position, rotation and scale. They can both be independent and influenced by the assigned mesh data.

Scene Data: Contains additional information of different objects within the scene like light vectors, camera position and distances to other objects.

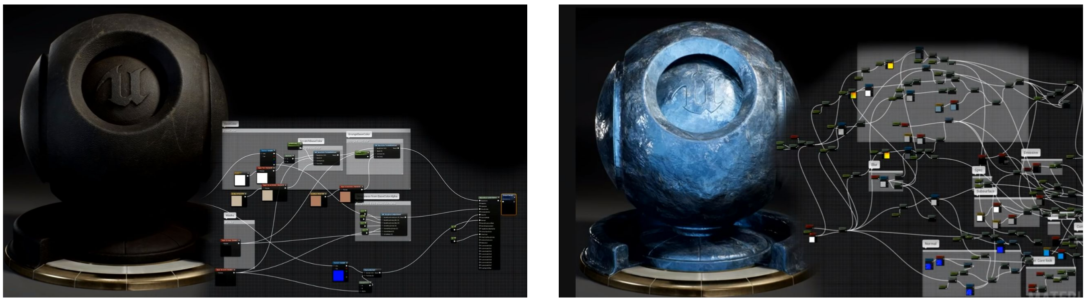
4.1.3 Material Layer

The material layer is a data type that describes the standardized material properties by defining the individual channel values influencing the corresponding render passes. This standardized format makes it easy to blend and process base material information in a generalized way. This input can be used to drive procedural processes within the masking or material container. In the context of *UE4*, this concept corresponds to *Material Attributes.*[4]

4.2 Material Container

This category describes the parts of the material layering system or shader graph that are used to define the surface properties of a base material. A material container is defined by different input data like textures, variables, mesh data, scene data or material layers and the computation happening within. The material container outputs a standardized material layer object. Figure 4.3 shows two base materials: a simple and a complex one. A common workflow is to use textures to define the influence of the material on the individual rendering passes. This is a proven approach and used across many productions. The material container performs limited to no computation at all, except combining those textures to a material layer object and pass it on. Another approach is to use a more procedural workflow. In this case the material creation can react and adopt

[4]See the article about *Material Attribute Expression* in the *UE4* documentation https://docs.unrealengine.com/en-us/Engine/Rendering/Materials/ExpressionReference/MaterialAttributes.



(a) (b)

Figure 4.3: Example of independent material containers in *UE4*. These materials are implemented as material functions in *UE4*. Both base materials use the same material parameter outputs but differ in complexity: (a) simple base material, (b) complex base material. Image source: [30].

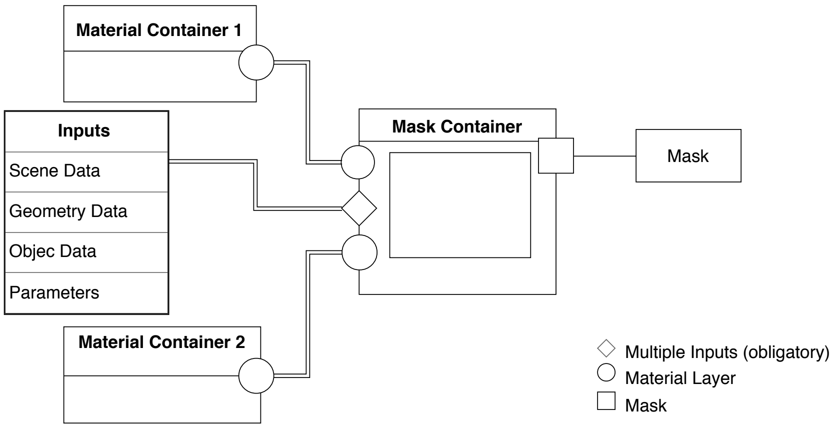


Figure 4.4: Masking Container.

dynamically to different input data. This input data is used by procedural instructions. This can be a position and height based gradient tint or time based offset of the UV coordinates. The possibilities are limitless. As these instructions are usually calculated at runtime, they do not use a lot of memory but rely heavily on computations. Using parametric inputs in contrast to textures is usually a trade-off between memory and computation time.

4.3 Masking Container

The masking container is responsible for controlling the affect area and intensity of the blending module. It provides the same full access to the different inputs as the material container. Nevertheless, texture masks are the most supported input type within material layering systems. Texture masks are an intuitive and straight forward approach of masking out areas. The value 1 (white) means that the texture is visible while the value 0 (black) means it is not. The values in-between allow partial blending between the inputs and therefore allow to mix different values—in this case different material container within the blending module. There are many different approaches to how to generate and project those masks onto a mesh. Some applications like *Substance Painter*, *DDO Painter*, *Blender*, *3D Coat*, *Mari* allow to paint masks directly onto the mesh itself. This provides a lot of control over the actual blending. Procedural processes or baked maps can also be involved in the creation process of these texture masks. It is important to keep in mind that the final texel density does not dependent on texture mask resolutions. The texture resolution does only influence the resolution of the blending, i.e., the transition between the affected material components. The higher the mask resolution is the more detailed the transition between the textures appears. A 1k mask can be used to blend two 4k textures. This means that the object texel density is sixteen times higher than the texel density of the mask. For optimization and organization reasons gray scale masks are often stored in different color channels within one single texture. These textures are often called id maps. Each color channel (red, green, blue, alpha[5]) represents another mask. The same rules as for the gray scale masks also apply to the ones within the color channels. An additional one can be stored within the alpha channel. Other commonly used inputs to influence the masking generation are vertex color, object and world position, object and wold space normal. Some of these inputs can be used to create different results even though the meshes are instantiated and shared. The different properties of these inputs can be used to achieve different behavior depending on the desired result.

The masking container can use and combine different inputs to compute the final mask output. Every input type has individual advantages and disadvantages. To get the most out of the blending, it is useful to combine different input types and combine texture based and procedural methods to generate the mask. Combining different inputs and methods can also be used to create additional detail for the main mask. This allows to either use methods that do not provide detailed masks (e.g., vertex color, world position, world space normals) or textures that have a low resolution. Additional details can be added later on by using additional inputs. An easy way to do so is to blend the main mask with a secondary tillable texture mask. This secondary mask is used to generate the small scale details. Using different blending modes (e.g., multiply, screen, add) gives a lot of control. Another less common approach is to use procedural noises as they are more expensive in terms of computation cost.

4.4 Blending Module

The blending module describes the part of the material layering model that is responsible for the blending of the different base materials. There are many different ways to do so. As mentioned in section 3.2.2, the blending of the individual material parameters is not trivial. The first part of this section describes some of the common issues that need to be taken into account when creating a layered material shader or using a material layering system. It discusses issues on an implementation level of blending individual parameters. The second part focuses more on a high level analysis of blending entire

[5]The alpha channel is expensive in terms of memory consumption. Using the alpha channel doubles the used memory for this texture as *UE4* changes the compression format from DXT1 to DXT5 [57].

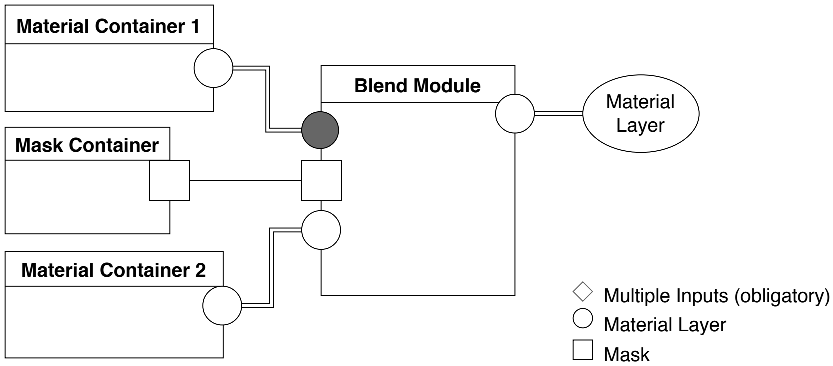


Figure 4.5: Blending Module.

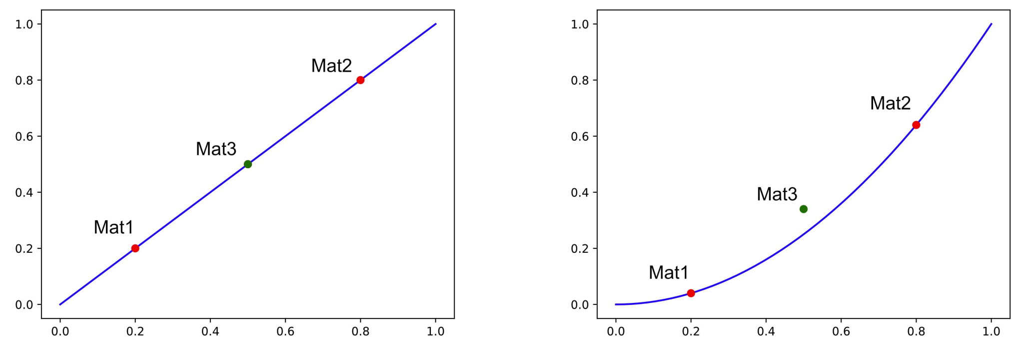
material containers.

4.4.1 Blending Individual Parameters

The blending of the different parameters can easily produce unintended values. One of these cases can appear by blending input parameters individually that affect the same material input. Pesare [63] provides a small example to further illustrate this point in his blogpost. By blending the colors A and B the average color should be achieved. Color A is gray with the rgb values (0.4, 0.4, 0.4), a diffuse gain of 0.5 and a resulting final color output of (0.2, 0.2, 0.2). Color B is a black with the rgb values (0, 0, 0), a color gain of 0 and resulting final color output of (0, 0, 0). The final color is hoped to be (0.1, 0.1, 0.1) as this is the average of the two final color outputs. What we get instead is a color with an rgb value of (0.2, 0.2, 0.2), a color gain of 0.25 and a final color output of (0.05, 0.05, 0.05)!

Another issue appears if the input parameter influences the output lobe in a non linear way. This is a common issue across different parameters. Commonly used algorithms to compute the specular lobes like Beckmann, GGX and GTR do not have a linear behavior between input and the size of the specular output. Therefore, the input values are not intuitive without remapping. They cause unexpected results when blending shiny with rough materials [15, p. 7–10 and 14–15]. This fact can easily be seen in figure 4.6. In practice, this issue is often taken care of by a careful shader design withing the 3rd party applications. The Disney *principled BRDF* and the physical shading models of *UE4* introduce a remapped quadratic roughness value to provide an approximately linear 0 and 1 roughness propagation [5, p. 15][15].

Another issue is the blending of two or more normal maps as they contain vector data. Simply adding or blending them does not bring the expected result. Jack Caron [48] does compare different approaches of blending normal maps on his website and provides some examples. To get a proper normal blending for the fire hydrant example in figure 3.1, I used the approach proposed by Christopher Dutton in *Correctly and accurately combining normal maps in 3D Engines* [6]. There are many different types of data that need to be blended within a pattern layering system and every single one has to be treated differently. The effort to ensure a proper blending from an artist or



(a) (b)

Figure 4.6: Blending material properties with not linear value propagation. This can be caused by the algorithm or multiple values influencing the same output lobe. In figure (a) we can see that blending the inputs from *Mat1* and *Mat2* results in the expected output. In figure (b), on the other hand, the output result of linearly interpolating between value *Mat1* and *Mat2* results in a wrong value that is not on the output function.

tool's perspective is huge.

4.4.2 Blending Modes

The blending modes for physically based material layering are just a few. There are only a few possible ways how materials are combined in the real world. They are either mixed together, stacked on top of one another or transition from one material into an other. Recreating real world behavior is a common approach to build systems that work intuitively and consistently for the user. Weidlich [22] explains that *Weta Digital* uses a material layering system that tries to be highly physically accurate. The blending modes for the material layering system are based on real world examples. *Weta Digital* includes two additional not physically plausible blending modes: one exclusively for adding emission and the other, rarely used one, for some specific exceptions. The following list contains the blending modes supported by the material layering system at *Weta Digital*, which is an example for a physically plausible system [22]:

Mix BxDFs: Two substances are mixed and the physical properties get combined (e.g., ink in water).

Coating: This method simulates the complex scattering process of light passing different thin material layers (e.g., water on ground, car paint).

Blending: Blending allows a transition between different base materials.

Adding: The adding of materials is only used for emissive materials as it breaks the energy conversation law and therefore is not physically plausible. Emissive materials break the energy conservation law anyway by introducing additional energy to the scene. It is therefore not relevant for them.

Subtract: Subtract does not correspond to any physical plausible combination of materials and is only used in few special cases.

My first attempt to create layered materials for the project *Letzte Worte* was to adopted this concept of using only a few physically plausible blending modes. During my research I found examples that use a much more flexible way of blending materials. In *Paragon* [37] they use the same normal maps for several base materials and blend only certain material properties to create material variation. This shows that the demands to a blending module vary depending on its application. It makes sense to use limited physically plausable layer blending modes in the context of a highly physically based rendering process. Whereas giving the user more control over the blending process, allows for necessary performance optimization in high quality games. Texture count and computation times can be reduced by allowing custom blending modes with control over which material attributes are to blended and how. This represent a trade-off between accuracy, usability and performance. Some material containers in *Paragon* are only described by a few parameters, instead of using a full material description on every material container. Instead of blending all material attributes, only the most important ones are blended. This decreases physically accuracy but increases performance dramatically. A powerful material layering system for real-time application should therefore include different options of how to define different materials and how to blend them. In the end, it is often a balancing act between physical accuracy, artistic freedom and performance. Allowing the user to control if, how and which channels are supposed to be blended puts the power and responsibility into the hand of the user. It provides an active choice between performance, usability and accuracy.

UE4 offers different nodes to blend material containers, override specific material layer attributes or get them individually. A list of these nodes can be seen in table 4.1. They take care of proper blending of individual parameters and therefore prevent users from tapping into issues correlated with pattern layering, discussed previously in section 4.4.1. It even offers different blending possibilities, e.g., the simple *MaterialBlendNode* sacrifices accuracy for performance in comparison to the standard one.

4.5 Summary

In this chapter I introduced a schematic model to describe arbitrary material layering systems or shaders, the material layering model. It consist of three components: the material container, masking container and blending module. A material container has arbitrary inputs defined in the shader or layering system and outputs a material layer object. The material layer object contains all information necessary to describe the rendering properties of a surface. The masking container can also access arbitrary inputs defined by shader or layering system. It outputs a 0 to 1 mask that is passed on to the blending module. The blending module is responsible for blending the material containers. The intensity and area of effect are defined by the mask.

Moreover, this chapter discussed shader inputs as they are an inherent part of both material container and masking container. To use and combine the distinctive properties of the individual inputs individually increases the possibilities of a material layering system. Finally, the blending modes and processes within the blending module were discussed. When blending material containers, the propagation, data type and input of individual parameters have to be considered. To simply interpolate linearly between different normal, roughness or ior values can produce unintended results. Another point

Table 4.1: Different Layered Material Blend Types in *UE4*. Source: [52].

Material Blend Mode	Description
AO	Blends an ambient occlusion (AO) map over the surface to remove reflection.
BaseColorOverride	Allows the Base Color to be replaced.
BreakBaseColor	Outputs the Base Color from an incoming Material Layer.
BreakNormal	Outputs the Normal from an incoming Material Layer.
Decal	Blends a decal sheet over the Material using the 2nd UV channel.
Decal_UV3	Blends a decal sheet over the Material Layer using the 3rd UV channel.
Emissive	Blends an Emissive texture over the Material Layer.
GlobalNormal	Blends a Normal texture over the Material Layer.
LightmassReplace	Replaces the Base Color in Lightmass, allowing for changes to indirect lighting results.
ModulateRoughness	Multiplies the Material Layer's Roughness by an incoming texture. Useful for a "greasy" look.
NormalBlend	Blends a Normal texture across the surface, but by way of a mask texture, allowing for control of where the normal will appear.
NormalFlatten	Diminishes the effect of the Normal map.
RoughnessOverride	Replaces the Roughness texture of a Material Layer.
Simple	Provides a simple linear interpolation (Lerp) blending solution for 2 Material Layers. Does not blend Normal; instead, retains Normal of the Base Material.
Stain	Blends the Top Material over the Base Material as a stain, meaning that only the Base Color and Roughness values from the Top Material are used.
Standard	Blends all attributes of two Material Layers.
Tint	Allows for tinting of a Material Layer by inputting a tint color and a mask to control the tint's location. Useful for making partial color changes.
TintAllChannels	Similar to Tint, but also affects Specular. This is a very special case function; generally, you will not need it.
TopNormal	Blends all attributes of both Materials but only uses the Normal of the Top Material.

to consider is how much control and freedom is given to the user. Should the blending module enforce accurate blending or concede the freedom to customize and optimize the blending process. The next chapter will put these abstract concepts into practice and will focus on the implementation of material layering methods into *Unity* and *UE4*.

Chapter 5

Material Layering in Unreal and Unity

UE4 as well as *Unity* include different approaches on how to work with material layering. The chapter starts with an introduction of how *Epic* designed their node based shading model for *UE4* to work for material layering. This is an interesting point if you consider implementing a shading model yourself or if you want to get a better understanding of shading models in other software. The chapter continues by explaining which systems already exist in order to create layered materials within the engines, how to access them and how they work. The use cases, advantages and disadvantages of the individual systems are described later in the design patterns category (see chapter 7).

5.1 *UE4* A Shading System for Material Layering

In *UE4*, the ability to create a simple and efficient material layering system was an essential requirement when recreating the shading model for the *UE4*. This shading system uses a simplified and adopted model of the *Disney's* principled BRDF [5] to fit the requirements of a real-time engine [15, p. 9]. The unification of the shader model allows the recreation of most real world materials by sharing the same parameters across all shaders. This is a fundamental point for creating an efficient pattern layering workflow as already mentioned in section 3.2.2. The *UE4* developer removed some advanced shading techniques (e.g., subsurface scattering, transparency, clear coat) from the standard shading model. For optimization reasons, they implemented individual shader models for those special cases. They simply expanded them by some additional features. As these shading models are almost identical, it is easy to change the shading model of a shader later on.

Adaptations of the principled BRDF shading system are widely implemented in a vast variety of other 3D packages such as the *Substance Suite* [60, p. 8], *Blender* [47], *Marmoset Toolbag* and many more. Using similar input parameters for the shading system makes a production pipeline between different softwares possible. Materials can be previewed, created and manipulated in other softwares with minor changes in the material inputs (e.g., textures and vertex attributes). Due to minor differences in the implementation, it might be necessary to adapt the textures and material inputs to achieve identical visual results. The *Substance Suite* includes presets to export the textures properly for the target applications like *UE4*. All modifications are stored in presets and can therefore simply be exported when exporting the textures.

One simplification done in the shading model of *UE4* was to treat advanced shader instructions like subsurface scattering, anisotropy, clearcoat and sheen as special cases, separately from the standard shading model. This was done to minimize the performance overhead. The base shading model for *UE4* contains base color, metallic, roughness and cavity [15, p. 9–10]. To the best of my knowledge, the cavaty parameter is not an explicit input parameter in the *UE4* shader but rather a generated cavity map based on the the normal map. This cavity map is used to create small scale shadows that could not be produced by regular real-time shadows. This cavity map is then multiplied onto the base color and the specular value. The specular value is set to 0.5 by default which represents a constant f0[1] value of 0.04 for non dielectric materials [54]. This value can be overwritten by the specular input.

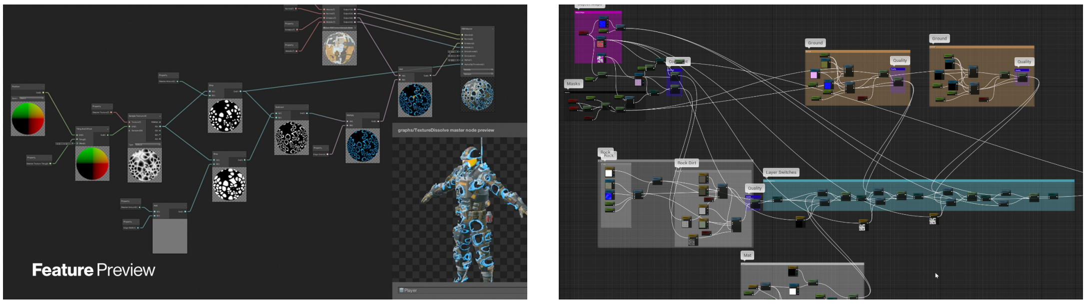
5.2 Material Layering Implementation

Both game engines—*UE4* as well as *Unity*—support different ways of working with material layering. One method provided by both engines is to create custom shaders using the shading language Cg/HLSL [55, 66]. Writing custom shader code requires much experience and time but provides the most control. Additionally, both engines offer other tools to customize shaders, for example the shader graph editors. They provide access to a huge library of shader operation and input nodes. The system automates a lot of the optimization work and creates automatically optimized shader code in the background. Working in a node based workflow makes creating custom shaders more accessible and user-friendly. This allows even less experienced users with less technical background to create complex shaders. *UE4* provides two additional systems to support and streamline material layering. These systems provide a workflow to increase re-usability, flexibility and efficiency. They are mainly workflow systems. Identical visual results could also be achieved by using any of the other methods mentioned here. Figure 5.2 shows a simple example that was recreated identically by using different approaches.

5.2.1 Node Based Shader Graphs

Both real-time engines—*Unity* and *UE4*—provide a node based way of authoring shaders (see figure 5.1). These powerful tools provide access to a huge amount of shader operations, functions and inputs that can be used to create custom shaders as well as layered material shaders. Both systems are flexible enough to create layered material shaders using several base materials and complex masking and blending operations. The node based design makes it easy and fast to iterate on shaders and adopt them constantly to the project needs. All examples and tests were performed by using these shader graphs and the specific layered material systems built on top of them (see section 5.2.2 and section 5.2.3).

[1] Fresnel zero defines the percentage of specular light that is reflected on a surface directly facing the camera [60, p. 10].



(a) (b)

Figure 5.1: *Unitys* shader graph editor (a) and *UE4* material editor (b). Both engines, *Unity* and *UE4*, include a node based editor to create and manipulate custom shaders. Image source: [49].

5.2.2 *UE4* Material Layering V1

The first material layering system was implemented as an extension of the pre-existing material function[2] system [52]. The different parameters (base color, roughness, metallic, etc.) of a base material—a material container—can be combined into one node by using the *Make Material Attributes* node. This can be thought of as a container including all parameters necessary to describe the surface properties of a material. All parameters of the base material are now handled as a single output. This material container or layer can also be easily moved out into a self contained sub-graph (a material function) and be referenced into any other shader graph. The architecture of these material functions allows reusing them as many times as wanted. This works for all kinds of different shaders. Single material layers can be blended together in different ways by using pre-defined *Material Layer Blend* functions. They provide different blending modes and operations for splitting, combining and manipulating single parameters. Section 4.4 discusses the issue of blending single material attributes. It also shows how material layering system can support the artist in blending them properly. In one of *Epic's* live streams introducing the newer material layering system [25], Alan Willard, Lauren Ridge and Chris Bunner summarize the most important shortcomings of the material layering system v1:

Clarity: Creating a master material using many different layers can end up in a huge graph which might be difficult to understand at a later point in the project. This issue becomes more urgent the more layers are added and the more material instances and different combinations of layers are needed.

Flexibility: Adding a new layer or layer variation for certain objects result in the need to update the shader graph. This can either be done by using static switches or adding the changes to a new copy of the shader. The former results in a growing node tree. Every change in the shader graph results in a re-evaluation of all material instances. The latter results in big number of different shaders. Changes and

[2] Material functions are parts of a shader graph that can be saved as independent sub-graphs. They can contain complex shader graph networks. Material functions can be reused across different shader graphs, stored in libraries and shared amongst artists.

optimizations are not automatically propagated to all materials and need to be added manually to all shader graphs if not contained within a material function. A growing complexity within the shader graph of an increasing number of shader graphs decreases flexibility and simplicity.

Functions: The material layering system relies on material functions and therefore shares the issue of this system, like the difficulty in passing on parameters. Parameters need to be explicitly exposed from the material function to be accessible from the surrounding shader. Parameters cannot be exposed directly from the material function to the parameter windows of the material instances.

5.2.3 *UE4* Material Layering V2

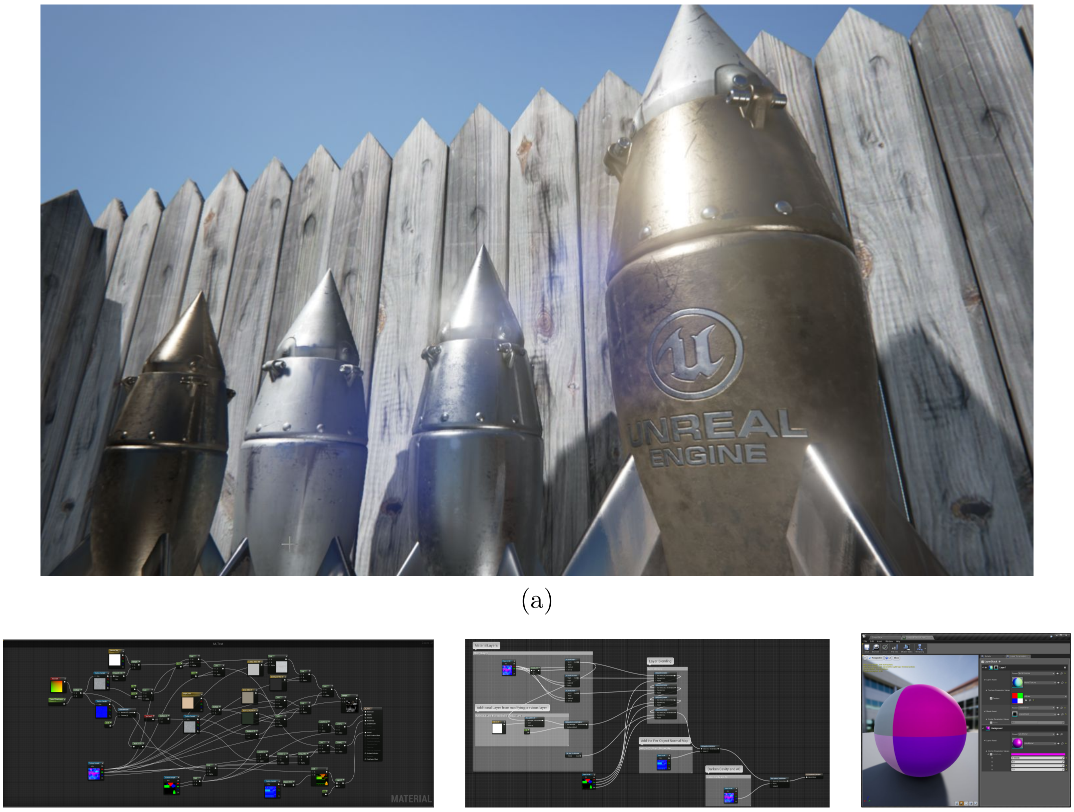
The new material layering system represents an entirely new workflow. The same results could be achieved with prior methods, but ease of use, flexibility and clarity have been improved hugely. The material layering system has been streamlined and split into different components: the *Material Layers* and *Material Layer Blends* [53]. This represent a logical separation between defining a layer and blending the independent layers. A *Material Layer* is a material container defining a base material using an independent shader graph. *Material Layer Blends* handles the blending between two arbitrary input materials by defining masking as well as blending. The architecture of the *Material Layering V2* is similar to the schematic high level description presented in chapter 4. A *Material Layer* object in *UE4* is equal to the concept of a material container. The *Material Layer Blend* includes both the blending module as well as the masking container.

In the old system, replacing a base material, cobblestone ground with concrete or simply adding a new layer, would result in the need to add an additional base material into the shader graph. As mentioned before, this introduces new switches and a growing shader graph. Alternatively, a modified copy of the shader could be created. Anyway, this results in an increasingly complex shader graph and the re-evaluation of all material instances using this shader. In the new system a base material layer can simply be swapped out without any need for manual change or the re-evaluation of all material instances. This is possible because the material containers are not specified by the shader. The material layering system creates the shader dynamically based on the used *Material Layers* and *Material Layer Blend*s. This logical separation of layer definition and layer blending eliminates most of the former shortcomings mentioned in the section before, such as flexibility, clarity and the need to re-evaluate all material instances.

5.2.4 Pre-existing Layered Material Shaders

Alegorithmic—a software developer specialized on texture authoring tools—provides a layered material shader for *Unity* and *UE4* that can be downloaded from *Substance Share* website.[3] This shader allows the blending of five or ten base materials that are all defined exclusively by texture inputs. Additionally, it supports objects specific textures like normal map and ambient occlussion. These shaders are not a blackbox. In both

[3]Download for *UE4* https://share.allegorithmic.com/libraries/2125 and *Unity* https://share.allegorithmic.com/libraries/2126).



(a) (b) (c) (d)

Figure 5.2: Identical visual result from different pattern layering systems. Figure (a) shows the final object using material layering. Figure (b) shows a shader graph setup without using material layers. The example on figure (c) shows the shader graph using the *Material Layering V1*. As comparison, figure (d) shows the UI for the the *Material Layering V2* system. Image sources: [53, 63].

game engines *Unity* and *UE4* the shader code or the shader graph setup can be modified. Hugely modified versions of this material were used for some of the layered shaders in the project *Letzte Worte*.

Unity gets shipped with a layered shader called *LayeredLit* as a part of the *HDRenderPipeline* package. This shader is designed for environment assets created by using photogrammetry. Nevertheless, this shader could also be used for any other kind of asset. Creating an efficient, flexible and easy to use shader affords much time, expertise and testing. Using this pre-existing layered shader can therefore be really useful. The shader code is included in the *HDRenderPipeline* package and can easily be adopted and modified to fit the specific project needs.

5.3 Summary

An interesting aspect of this chapter is how *Epic* already considered material layering when redesigning its node base shading graph system for *UE4*. An important step hereby was to standardize the shading model. It uses the same shader parameters across all materials. For performance optimizations they introduced variations of the shading model supporting advanced features (e.g., subsurface scattering, translucence, clear coat). The individual shaders can easily be adopted to fit another shading model. The second important step was to use algorithms that propagate in a linear and proportional way. Especially as regards the roughness value, this solves the issue of wrong roughness accumulation on an engine level. *Unity* as well as *UE4* implement different methods to use material layering. Both enginges support custom shaders—written in Cg/HLSL—and powerful in engine shader graph editors. On top of this node based shading system, *UE4* implemented two dedicated systems to improve the material layering workflow. Before passing on to material layering design patterns, I want to examine the approaches of other authors to develop such catalogs for decision making.

Chapter 6

Design Patterns

The previous chapters mainly focused on technical aspects of material layering. This chapter investigates the interdisciplinary concept of design patterns. The purpose and goal of design patterns is described best by Christopher Alexander [2, p. 10]:

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Christopher Alexander et al. [2, 3] had a huge impact on many different disciplines. Their books introduce a network of interlinked patterns. The patterns form a language provided in a standardized format. This makes the individual patterns easily understandable. The purpose is to offer pragmatic solution to recurring problems. The patterns should give an explanation for how and why to use them. They make it easy to evaluate, analyze and modify them for the specific situation. Besides, they are interconnected and reference one another. Patterns are seen as modules of a language that can easily be combined and cannot exist as isolated entities [2, p. 10–13]. Although this concept of patterns was initially developed for architecture, it was adopted in many other disciplines.

Another subject that adopted design patterns is software engineering. Erich Gamma et al. [10] introduced a catalog of proven design patterns for object-oriented software engineering. This book contains a collection of solutions that evolved over time and could be applied to all kinds of applications and projects. *Game Programming Patterns* by Robert [18] provides additional design patterns especially—but not only—for game programming. Eric Freeman et al. [9] try to equip the readers with a more accessible and practical insight into design patterns for object oriented programming. In contrast to [10], it is more a step by step guide to different design patterns rather than a catalog. Jenifer Tidwell [21] established design patterns for Interfaces in her book *Designing Interfaces: Patterns for Effective Interaction Design.* This publication provides different possibilities for creating UI elements and how to use them. These are just a few examples of disciplines that adopted the idea of design patterns.

Method	*Status*
DP [Number]: Pattern Name:	Unique number and name of the design pattern (DP).
Intent	A brief statement of how the pattern works and the issue it is supposed to fix.
Also Known As	Is this approach known by other names as well?
Motivation	Illustrates use cases by naming some practical scenarios.
Applicability	Provides conditions where this pattern is appropriate.
Implementation	Do different applications implement this pattern and how well?
Examples	Showcases different examples from real projects.
Consequences	What are the consequences of using this pattern?
Related Patterns	Other pattern that are connected or dependent on this one.

Table 6.1: Design Pattern Template.

6.1 Design Patterns Structure

A really important part of defining design patterns is to find a way to organize them. As seen before, design patterns are not supposed to be used as isolated, single solutions, but as a part of a flexible and connected language. Christopher Alexander et al. [2] organize the patterns in a sequence, going from big to small, from urban planning to small architectural detail. Each of the patterns is connected to others with regard to larger and smaller scale categories [2, p. 10–36]. Erich Gamma et al. [10] adopt a rather functional approach for categorizing the design patterns which are defined by their purpose (creational, structural or behavioral) and scope (classes or objects). Jenifer Tidwell [21] uses thematic chapters to organize the patterns like *organizing the content*, *organizing the page*, *showcasing complex data* and many more. The overall structure for this work can be found in the next chapter 7.1. The individual design patterns presented in this work use a uniform template similar to the ones from Erich Gamma et al. [10, p. 16–18] and Jenifer Tidwell [21, p. 42–46]. This standardized format is supposed to make it easier to extract important information, compare different design patterns and take decisions. The template consists of following points presented in table 6.1.

Another important fact I want to point out here relates to the aspects these patterns do not represent. These points are inspired by Jenifer Tidwell [21, p. 42–46]. The following patterns are neither fundamental principles nor focused on a specific implementation. They are abstract enough to work in a variety of situations. They might be adopted slightly to fit the special purpose. Some of the patterns might work across different engines, applications and even for film productions; others might not. These patterns are proposals. It is the user's decision if these proposals make sense for the specific project and pipeline. A pattern describes a relation between different components. For instance, using a texture mask is not a design pattern at its own, but how and when to use texture masks in the context of a blending Module is one.

6.2 Summary

As described in this chapter, design patterns provide solutions to recurring problems in a standardized form. The structure or the patterns in the following chapter is inspired by other industries. They form a language of many interlinked objects. The template for the patterns has been defined and contains all the important information. The standardized format makes it easy to compare different patterns and choose between them. Finally, this chapter defined what design patterns are not: they are neither high level abstract principles nor do they define specific low level implementations.

Chapter 7

Design Patterns for Pattern Layering

This chapter contains a catalog of design patterns related to material layering. The pattern catalog presents the core of this work and incorporates all previous results and analysis. It combines the results of the previous research and presents them in practical patterns. The patterns are categorized and structured in a hierarchical way. This is supposed to simplify the process of finding design patterns related to the actual problem. The patterns start from high level decisions of how and if to use pattern layering and get more and more specific towards the end. Figure 7.1 shows the relationship and categorization of the design patterns. A list of all patterns can also be found in appendix A.

The pattern catalog starts with the most fundamental questions: Is a pattern layering system the best solution for the particular asset, project and pipeline? Pattern *DP 01: Pattern Layering* (section 7.1.1) and *DP 02: Hybrid Pattern Layering* (section 7.1.2) illustrate the advantages and disadvantages of this workflow. Alternatives to pattern layering are proposed in category *Alternatives*. Some of the most important alternatives are further elaborated in patterns *DP 03: Baked Texture Maps* (section 7.2.1) and *DP 04: Different Materials* (section 7.2.2). The second big question is how to incorporate pattern layering in a specific project. Resources and expectations differ tremendously between projects. The requirements in a pattern layering system for a *AAA* game are completely different from those of a small indie team. The patterns *DP 05: Built-in Shader* (section 7.3.1.1) and *DP 06: Custom Shader* (section 7.3.1.2) discuss the best way how to utilize pattern layering in a specific project. This is closely connected to how the shaders are supposed to be organized. The patterns *DP 07: Uber Shader* (section 7.3.2.1), *DP 08: Individual Shader* (section 7.3.2.2) and *DP 09: Content Generated Shader* (section 7.3.2.3) present options on how to organize and structure them. The material layering model discussed in chapter 4 provides the foundation for splitting pattern layering into its sub-components. The first module is discussed in category *Material Container*. A fundamental decision is how granular the individual base materials are represented. Patterns *DP 10: Base Material* (section 7.4.1.1) and *DP 11: Material Variation* (section 7.4.1.2) discuss the approaches of either using a bigger number of generic base materials or a smaller amount of more complex ones. This can be connected to the patterns *DP 12: Full Material* (section 7.4.2.1) and *DP 13: Modulation Layer* (section 7.4.2.2) and the decision of how complex individual base materials should be. Finally, patterns *DP 14: Input Based* (section 7.4.3.1) and *DP 15: Semi Procedural* (section 7.4.3.2) discuss different ways of defining the surface properties of a material container. After deciding on how to create and structure those base materials, the question of how to blend and mask them arises. The category *Masking Container* is focused on how to create the blending masks. The decision to either use texture based (see *DP 16: Texture Based* in section 7.5.1) or procedural inputs (see *DP 17: Procedural* in section 7.5.2) influences the workflow dramatically. Patterns *DP 18: Physical Material Blend* (section 7.6.1) and *DP 19: Custom Material Blend* (section 7.6.2) present different approaches to how to handle the blending of different material containers and how to balance realism, usability and performance within the blending?

Finally, the category *External Inputs* discusses different shader inputs further. External refers to data that is not embedded within the shader. This data can be used to drive computation in the material container or masking container. Parameters are one fundamental input type. Patterns *DP 20: Textures* (section 7.7.1.1), *DP 21: Variables* (section 7.7.1.2) and *DP 22: Scripted Parameters* (section 7.7.1.2) demonstrate how these parameters can be used. Properties differ depending on input type. Patterns *DP 23: UV Coordinates* (section 7.7.2.1) and *DP 24: Vertex Color* (section 7.7.2.2) demonstrate how mesh data can be used to achieve different results. Patterns *DP 25: Vectors* (section 7.7.3.1) and *DP 24: Vertex Color* (section 7.7.2.2) discuss object related input data.

7.1 Pattern Layering

This catalog starts off with two different approaches to include pattern layering into your workflow. Patterns *DP 01: Pattern Layering* and *DP 02: Hybrid Pattern Layering* present two different approaches to do so. The former introduces a pure pattern layering shading. The latter proposes a hybrid method that combines pattern layering with more traditional object specific approaches. These patterns in connection with the patterns in category *Alternatives* will support you in taking the biggest decision connected with this pattern catalog. Does pattern layering make sense for your specific project? The second question that arises is whether pattern layering alone provides the desired result or is a hybrid approach better suited.

7.1.1 DP 01: Pattern Layering

Intent: Use different generic and tileable material layers and combine them to recreate the visual appearance of complex surfaces. Built a pipeline based on re-usable components and tools (e.g., material library and parametric masking).

Also Known As: *Parametric Layering*, *Material Masking System (MMS)*, *Layered Materials*, *Material Layering*, *Dynamic Material Layering*

Motivation: Consider working on a huge scene for a first person shooter. Most areas can be seen from both a further distance as well as from really close up. Remember the castle wall example from chapter 1 that was composed of many different base materials. Working with unique pre-baked texture maps would largely exceed the memory if trying to achieve a consistent high texel density.

The asset and shading pipeline of the company or team you work for transitions more and more into the engine. Different artists use different tools for the textur-

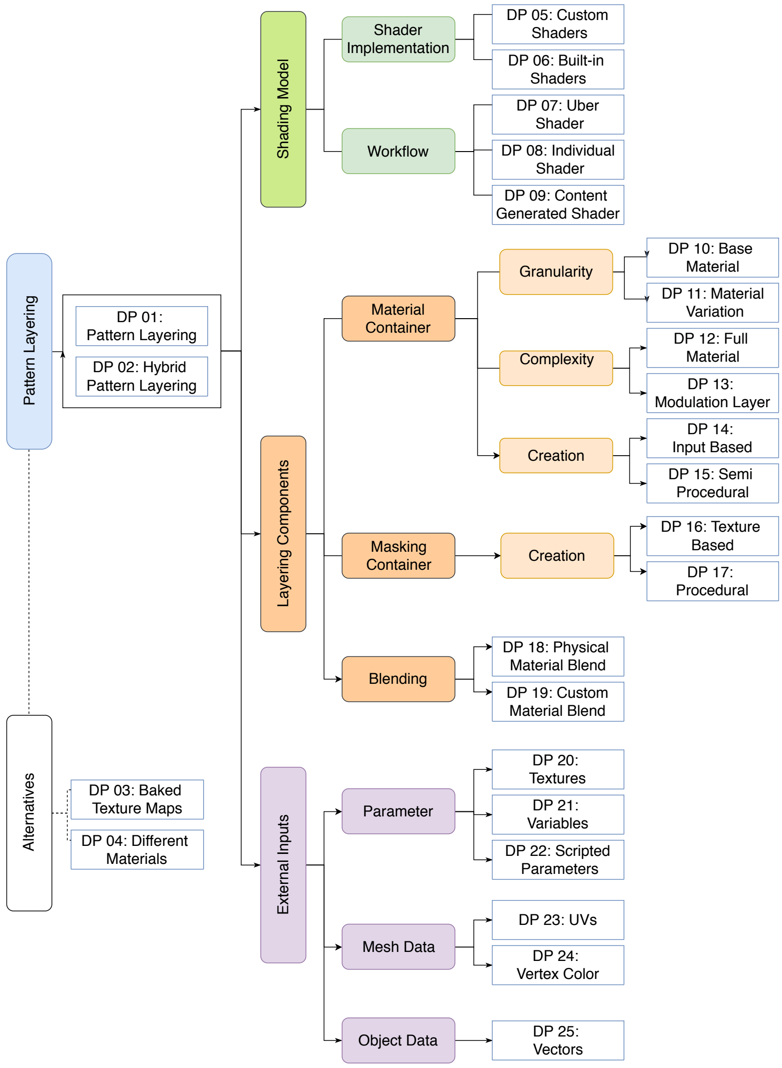


Figure 7.1: The hierarchical structure of the design patterns presented in this work. The primary element is *Pattern Layering*. The category *Alternatives* shares the same hierarchical level as it proposes equally valid alternatives. All of the other elements are descendants of *Pattern Layering* (*DP 01: Pattern Layering* and *DP 02: Hybrid Pattern Layering*). The categories *Shading Model*, *Material Container*, *Masking Container*,*Blending Module* and *External Inputs* contain more specific and detailed patterns and progress from high level workflow and organization decisions to specific low level features.

ing. The companies goal is to unify the material pipeline, create a central base material library and use advanced shading instructions (e.g., real-time tessellation, parallax occlusion mapping and vertex shader animation). These advanced shading techniques cannot be reproduced and shared across different applications easily.

Applicability:

- All base materials use the same shading model.

- The shading system has to account for a huge, complex surface area where different base materials are used. The transition of these base material is smooth, geometry independent and most likely on a per pixel basis.
- The blending uses procedural components that respect any properties defined in the material container. For instance, both base materials are blended according to their height map so that base material B appears only in the valleys of base material A.
- The masking and blending of the layers uses various information from the scene, such as position, normal direction and so on.
- The blending of different base materials is modified dynamically at runtime. Imagine peddles integrated into the material shader that respond to global weathering systems within the game.
- A lot of different material variations of the same object are needed.
- The workflow is supposed to be iterative. Individual base materials can change independently and are easy to be replace if desired.
- Base materials can be used and re-used across different objects, projects and artists.
- The project uses a centralized art directed material library that can be shared across different artist. Changes on the base materials are propagated down to all assets using them. Working with a centralized art directed material library also improves consistency across different assets.

Implementation: There are many different ways to implement a pattern layering (see chapter 5). The user experience differs strongly according to which one is used. The different approaches to implement a pattern layering system are to use either built-in shaders (*DP 05: Built-in Shader* in section 7.3.1.1) or create custom ones (*DP 06: Custom Shader* in section 7.3.1.2). Custom shaders can be implemented using HLSL/Cg, shader graph editors or material layering systems.

Examples: Most of the patterns presented in this catalog were tested within the VR experience *Letzte Worte*. A lot of these patterns arose from questions I had myself throughout the project. The suggestions found in different publications, talks, articles and sample projects helped me take decisions on how to handle an individual problem. All these practical experiences flowed back into these patterns. In *Letzte Worte* I used pattern layering for all environment assets except detail props like electronics, machines, books, lamps etc. *Letzte Worte* uses a wide variety of different patterns presented in this catalog.

Consequences:

Visual Qualities:

- Pattern layering forces the artist to approach texturing in a technical way.
- It allows the shading of huge interesting surface areas while still achieving lot of surface variety. By using repeating tillable base materials, this method enables a high screen-space texture resolution while obtaining huge control over the blending.

- The artistic process is limited by the layered material shader. The shader defines a big part of the visual quality. A lot of artistic choices are taken while creating it.
- A lot of the artistic process takes place on a shader level. To achieve the best results, a good technical knowledge is needed.
- The approach to texturing is more technical than using 3D painting applications.
- Using a base material library supports a good art directability. All artists use the same materials. Changes on the base materials are propagated down to all assets using them.

Performance:

- Using a layered material shader can have a big impact on performance, especially depending on the complexity of the individual material layers. The real-time rendering engine computes all layers for all pixels simultaneously and blends them afterwards. Even base materials that are not visible for a certain pixel are computed [52].
- Texture memory can be decreased dramatically compared to traditional alternatives, as for instance *DP 03: Baked Texture Maps* (section 7.2.1). This is caused by the fact that texture maps can be used across different materials and the individual texture resolution can be much smaller as it is repeated multiple times over the surface area.

Pipeline (Workflow):

- It is hardly possible to exchange shading data with other applications. To do so often requires other custom pipeline tools.
- Blending individual material parameters of two material layers is not a trivial task. Either the pipeline or the artists have to account for the complex process of blending (e.g., roughness, ior accumulation). Please refer to section 4.4.1.
- Using custom shaders may require project specific tools and systems to support an efficient workflow. See pattern *DP 06: Custom Shader* in section 7.3.1.2.
- Creating a pipeline for material layering is time consuming. The development and testing of shaders and pipeline tools are expensive and time consuming.
- Working on *Letzte Worte*, I realized that using a highly customized shading workflow may cause issues with different features provided by the engine (e.g., hierarchical LODs and lightmaps).

Related Patterns: This pattern represents a pure pattern layering based shading workflow. *DP 02: Hybrid Pattern Layering* (section 7.1.2) proposes a more flexible approach that includes features of other methods as well. The category *Alternatives* contains alternative texturing approaches. Categories *Shading Model* (see section 7.3), *Material Container* (see section 7.4), *Masking Container* (see section 7.5), *Blending Module* (see section 7.6), and *External Inputs* (see section 7.7) con-

Figure 7.2: Different skins of *Chrunch*, a hero character from the *MOBA Paragon*. Pattern layering was used to quickly iterate and create variations of different skins during the production. This rendering was made by using the assets from the official *Epic Marketplace* [39].

tain further information on how to use and implement pattern layering in your project.

7.1.2 DP 02: Hybrid Pattern Layering

Intent: In order to create small scale details, an object specific single texture for the large scale variety and a more generic pattern layering workflow are to be combined.

Motivation: Photogrammetry is a process to capture meshes and textures from real world locations and objects. The data has to be optimized and cleaned for the use in a real-time application. However, it provides high quality with less effort compared to traditional methods. The advantages and disadvantages of using these textures directly are eplained in pattern *DP 03: Baked Texture Maps* (section 7.2.1). Matching the scanned data by purely layering base materials is difficult, time consuming and expensive. Even by matching the reference, it will most probably not be practical due to the huge amount of different layers and complex masks for blending them. Therefore, this design pattern combines aspects from both methods, i.e., object specific texture maps from pattern *DP 03: Baked Texture Maps* (section 7.2.1) with additional generic, tileable and reusable base materials as explained in pattern *DP 01: Pattern Layering* (section 7.1.1).

Applicability:

- The object fits all criteria for being shaded by using a pattern layering approach, but additional large scale information (e.g., photoscanned textures) is available.

- The artistic goal is not achieved by combining different generic base materials. Additional object specific large scale variety is needed.

Implementation: The hybrid pattern layering shares the same technical implementation as other layered material methods. In the context of the material layering model established in chapter 4, the additional object specific texture maps can be seen as individual arbitrary material container. On the implementation level, it can be treated like any other material container outputting an material layer object. Finally, different blending modes can be used to add to the final shading output.

Examples Figure 7.3 shows an example using pattern layering combined with object specific photogrammetry textures [58]. This example illustrates that combining real world captured object specific textures with pattern layering can be used to achieve realistic results. It further allows reusing and sharing tileable base materials across different assets. As shown in the image, this technique allows to easily create different variations and supports an iterative workflow.

Consequences Hybrid pattern layering is identical to *DP 01: Pattern Layering* (section 7.1.1) from a technical perspective. The additional object specific textures do provide some artist friendly possibility for adding large scale variety. These texture maps are not ideal to add small scale details as this would require huge texture resolutions and therefore eliminate the advantages of *DP 01: Pattern Layering* over *DP 03: Baked Texture Maps*.

Related Patterns This pattern combines aspects of *DP 01: Pattern Layering* (section 7.1.1) and more traditional methods like *DP 03: Baked Texture Maps* (section 7.2.1).

7.2 Alternatives

Before going over to how design and implement a pattern layering system, I want to show alternative approaches to texturing and shading your objects. The Patterns *DP 03: Baked Texture Maps* (section 7.2.1) and *DP 04: Different Materials* (section 7.2.2) contain workflows that are not necessarily associated with material layering. Nevertheless, they might be better suited for the particular requirements. Sometimes different approaches can be mixed as well (e.g., *DP 02: Hybrid Pattern Layering* in section 7.1.2).

Other material layering approaches that might gain importance in the future are BxDF layering (see section 3.2.3) and illumination lobe based layering (see section 3.2.4). To the best of my knowledge, pattern layering is the only material layering approach actively used for video game productions yet. Technologies like real-time ray tracing are most likely going to take over. They will change many workflows used nowadays; one of them will be the way how we handle material layering. I believe that BxDF layering (see section 3.2.3) and illumination lobe based layering (see section 3.2.4) will replace many use cases for pattern layering. This will be a huge game changer. As discussed in chapter 3, both techniques provide huge advantages as regards realism and usability. Another approach to realize material layering could be color layering (see section 3.2.1). This could be achieved by rendering multiple versions of the object and save the outputs in additional buffers. The data could be blended together in the post-processing stage. This

Figure 7.3: Combining 3D scanned texture data with smaller, generic and tileable base materials. This approach makes it possible to obtain the large scale texture variety captured from the real world object while ensuring a high texel density using Pattern Layering. Using pattern layering allows to easily and quickly create variations. Image source: [58, p. 6].

approach is expensive in computation and memory. Further experimental investigation needs to be done on this subject to see if there are any use cases for this approach.

7.2.1 DP 03: Baked Texture Maps

Intent: Use a pre-baked object specific single textures for the individual material inputs (e.g., base color, normal and roughness) to define the object appearance in the final render. Using object specific textures provides the most control over the final appearance on a per pixel basis.

Motivation: Consider an object with a huge variety of different base materials. The surface appearance is heterogeneous and it is really difficult to identify distinctive base materials. This makes it difficult to recognize recurring patterns in masks and base materials and recreate the surface by combining more generic base materials.

The specific asset plays a major role for either the gameplay or the plot of the game. Therefore, the most artistic control is required. You are working on small detail props for an office desk. These different assets (e.g., pencil, hole puncher, paper clip and post-it note) are composed of many different base materials. The list of base materials might look similar to this: rubber, plastic wood, painted wood, aluminum and paper. These different base materials do have additional variations. So for instance, there is a red and a blue pencil. The amount of base materials for texturing these assets with *DP 01: Pattern Layering* (section 7.1.1) is extremly high. As these assets are fairly small, they do not need a big texture resolution to achieve a high texel density. Additionally, this pattern is not limited by the amount of surface type variety in any way. It does not matter if a single texture represents one or fifty surface types.

Applicability:

- A low texture resolutions is enough to achieve a high screen-space texture resolution for the desired object. The necessary texture resolution does not necessarily refer to the dimensions within the 3D scene but to the maximum screen-space the object covers. A huge mountain located in a far distance can be textured by low res textures and still have sufficient screen-space texture resolution. A wall seen from close may exceed the screen size multiple times and therefore need a much higher texel density.
- The used base materials are specific to this particular object. The surface appearance cannot or hardly be recreated by combining more generic base materials.
- The surface area of this object is heterogeneous and composed of a huge amount of different surface materials. Splitting it into layers would result in a huge amount of different base materials and masks.
- This object has a special significance for the experience and the maximum amount of artistic control on a per pixel basis is required.

Implementation: Using baked texture maps is probably the most artist friendly shading workflow within 3D real-time applications. This workflow has existed for a long time, the tools and workflow have been highly simplified and optimized. This applies for both usability and performance. Most modern creation tools provide a workflow where you immediately see an output close to the final product. Artists coming from other fields can easily adapt to 3D painting tools, as they are structured similar to *Photoshop*. Especially since PBR has become a well established standard, textures for the individual material inputs can easily be shared and transfered throughout a pipeline. Content creation applications and real-time engines do offer pre-existing shaders that have already been set up. The user does only need to assign the textures to the proper material slot. Modern tools like *Substance Painter* eliminate much of prior shortcomings in texture painting applications. For instance, *Substance Painter* provides the possibility to simply increase the texture resolution even beyond the painting resolution. It enables the artist to paint and work on several textures simultaneously (e.g., base color and roughness). It provides pre-defined export presets to convert textures for the desired render engine, i.e., it adjusts the roughness curve to correspond to the specular algorithm used in the renderer.

Examples: *Letzte Worte* uses baked texture maps for all the detail assets. These assets are composed of many different surface types. Using pattern layering would result in an enormous amount of different base materials as well as their variations. These assets are small enough to achieve a high texel density even by using medium texture sizes. By combining a lot of these assets into one texture atlas and using textures compression, texture streaming and channel packing,[1] the memory usage is still highly efficient. Additional details that require even more detail (e.g., fonts, symbols, labels) use an additional UV set that provides the resolution where it is needed.

Consequences:

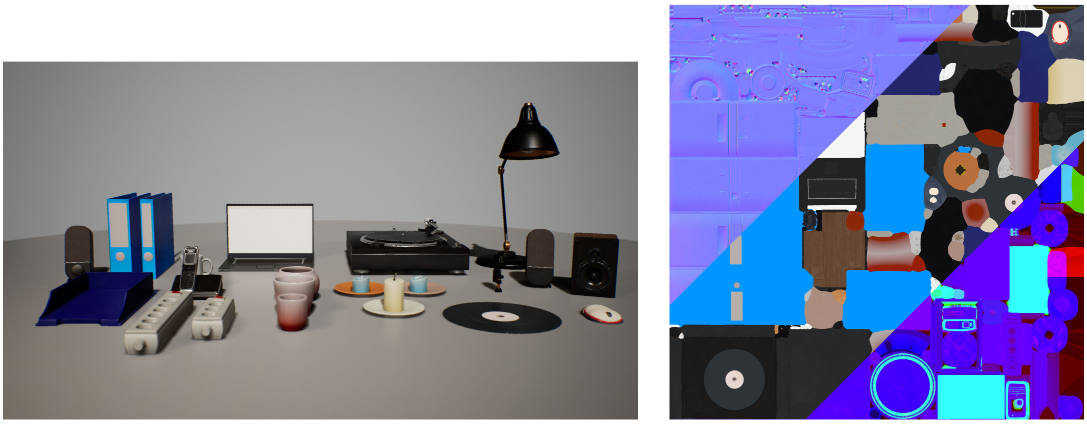
Visual Qualities:

- Using pre-baked textures provides the highest artistic control on a per pixel basis.
- It allows the use of an infinite amount of different surface types. There are no limitations on surface variety within the capabilities of the shader.
- Textures can be unique for every material.
- It is difficult to achieve a consistent art style across different artists and assets.

Performance:

- This method uses only a low amount of inputs. When packing different assets into texture atlases and using channel packing the amount of draw calls can be reduced.
- The engine provides many optimizations for handling textures like mid mapping, compression and texture streaming. The texture data gets only loaded when it is needed.
- The textures provide a lookup table containing complex surface descriptions. This information can be passed on to the shader without further

[1] Channel packing refers to the process of combining different textures into one by saving them into different color channels.



(a) (b)

Figure 7.4: Different assets using pre baked textures. The assets from figure (a) are from the game *Letzte Worte*. The textures are combined into a texture atlas to minimize draw calls. Figure (b) shows the texture atlases for the following maps: the normal map, base color and a channel packed texture (roughness, metalness and ambient occlusion).

computation. This keeps the shader complexity low even for complex surfaces.

- A big amount of high texture resolutions can cause a bottleneck on the VRAM which effects performance negatively.

Pipeline (Workflow):

- Baked textures are well integrated in different 3D applications.
- It is easy to transfer texture data across different applications. As most modern software uses a PBR system, textures can be shared quite easily.
- Textures are often object-specific, and therefore they can not be reused across different meshes.
- Changing the texture data means to re-export and import the maps from the texturing application to the game engine.
- A lot of production effort is put into guaranteeing a consistent quality and look across different assets.
- Creating object variations requires multiple textures.

Related Patterns: This pattern proposes an alternative to pattern layering, *DP 01: Pattern Layering* (section 7.1.1). A combination of both methods results in a hybrid form further explained in pattern *DP 02: Hybrid Pattern Layering* (section 7.1.2).

7.2.2 DP 04: Different Materials

Intent: Assign different materials to distinctive parts of the mesh to split the shading into individual sub-tasks. These isolated materials can be assigned to different

(a) (b)

Figure 7.5: Scene and assets from the *Infinity Blade: Grass Lands* map [36]. Figure (a) shows the final scene. Figure (b) shows a selection of used assets. These assets use the same materials, a tileable wall materials and a material using trim sheet textures. The latter is used to texture details like the columns and the railings. Image source: [36].

shaders and use different inputs. They are completely independent from one another.

Motivation: Consider a big architectural environment asset like a huge house. The object is composed of different distinctive areas like walls, roofs and windows. By splitting them into individual parts, material specific textures and shaders can be used. While the windows use transparency, the roof might incorporate fuzz shading for some mossy areas. Finally, the wall shader might use pattern layering and utilize vertex colors to provide more control over blending the distinctive base materials within the engine.

Applicability:

- The object can easily be divided into isolated areas representing different, distinctive surface types. The individual shader can utilize pattern layering or any other shading approach like pre-baked textures. The important requirement is that the separation between the materials uses a clear cut. It is not possible to create a smooth transition from one material to the other. By utilizing advanced shading techniques (e.g., world space UV coordinate, distance based normal interpolation or transparency), it is possible to make the transition appear smooth. Nevertheless, they are still two distinctive materials. Different materials are assigned on a per face basis; the mesh topology defines where a change in material can take place.
- Different parts of the object are highly different from a shading perspective and more flexibility in changing or modifying individual areas is needed.
- Generic, reusable materials can be used to define certain areas of the mesh. The material does not contain object specific data.
- The object is too complex or big to use one single material.
- The mesh uses different shading models for different areas, such as for example opaque, translucent and subsurface scattering.

Implementation: Different material IDs can easily be assigned to an object from within the 3D content creation application. The engine can use these IDs to assign engine materials to them. Guides on how to export and import meshes with corresponding material IDs can be found in the documentations[2] of both real-time engines.

Examples: Splitting objects into different materials is a common technique seen in almost any game. A particular example for utilizing different materials to shade a scene is the *2014 Soul* [27] demo, see figure 7.6. The location *Soul: City* can be downloaded from the *Unreal Marketplace.*[3] Figure 7.7 shows one of the environment assets. It uses 5 different materials, each to shade a distinctive surface type. These materials are stored in a material library and are reused across all the different city assets. Different city assets are merged together to further minimize draw calls. In this particular example, the material count stays similar as the assets use the same materials. Figure 7.7 shows one combined asset from the example files. By reusing the materials and merging the assets, the amount of render jobs sent to the GPU decreases dramatically. Instead of sending every material of every asset independently, all parts sharing the material get batched and sent together.

Consequences:

Visual Qualities:

- The assignment of different materials can only happen on a geometry basis. Materials cannot be blended. There are some advanced and expensive shading techniques that imitate the appearance of blended materials.
- Individual materials can use independent shaders and inputs.
- Materials defined by mesh independent inputs can be used across different objects and batched together for rendering, see figure 7.7.
- Splitting an object to use different materials may decrease the complexity of the shader.

Performance:

- Splitting objects to use several material may increases the number of materials and therefore the number of draw calls. Different materials are sent separately as individual draw calls to the GPU.
- An intelligent use of non mesh specific materials in combination with merged objects decreases draw calls.
- Splitting a problem into sub component might decrease shader complexity tremendously.
- It is generally more efficient to use several materials than using pattern layering, see *DP 01: Pattern Layering* in section 7.1.1. The *UE4* documentation states [52]: "In short, if you can apply multiple Materials instead of using a Layered Material, then do so. If you must have per-pixel control over where Materials are placed, then use a Layered Material."

[2] See https://docs.unrealengine.com/en-us/ for *UE4* and https://docs.unity3d.com/Manual/index.html for *Unity*.

[3] *Epic* published the map *Soul:City* [40] of their *2014 Soul demo* on the Unreal Marketplace (see https://www.unrealengine.com/marketplace/soul-city).

Figure 7.6: The *Soul: City* map [40] using different reusable materials for shading. The scene is highly optimized to run on mobile devices. Image source: [28].

Pipeline (Workflow):

- To exchange and replace individual materials within the engine is simple and fast.
- Changing the material assignment within the 3D content creation application is easy and fast. Re-exporting and importing are required though.
- The assignment of different material IDs cannot be done within the engine without custom tools.
- Individual materials can be modified independently. Changes are propagated to all meshes using the particular material.

Related Patterns: Splitting an object into individual materials to use different shaders and split the shading process into sub-tasks is universal to all methods proposed here: *DP 01: Pattern Layering* (section 7.1.1), *DP 02: Hybrid Pattern Layering* (section 7.1.2) and *DP 03: Baked Texture Maps* (section 7.2.1).

7.3 Shading Model

After deciding to use pattern layering for your project, the next big questions arise: How to implement and use pattern layering as well as how to structure and organize the shaders and materials. The following categories *Shader Implementation* and *Workflow* will provide patterns to help you making these high level decisions. They will help you to evaluate the benefits and constraints of using pre-existing shaders. What opportunities and dangers arise from creating a custom pattern layering shader or system? Is it better to create a centralized all purpose shader or problem specific custom shaders for my particular project?

(a) (b)

Figure 7.7: Different assets sharing generic, reusable materials. These renderings are made from *Soul: City* assets [40]. Figure (a) shows one of those assets. It is textured by using five distinctive mesh independent materials. They are reused across all city assets. Figure (b) shows different assets merged together. The material count increases only slightly. All mesh parts using the same material can now be batched for rendering.

7.3.1 Shader Implementation

Resources, budgets and requirements differ from project to project. Patterns *DP 05: Built-in Shader* (section 7.3.1.1) and *DP 06: Custom Shader* (section 7.3.1.2) propose different approaches to include pattern layering into your workflow. They discuss the benefits and limitations of pre-existing shaders and workflow as well as challenges when creating a custom setup. These patterns will help weigh up the two possibilities with regard to effort and benefit.

7.3.1.1 DP 05: Built-in Shader

Intent: Use pre-existing pattern layering solutions for your project and purpose. These take care of the complex blending of individual base materials but limit the amount of base materials, the kinds of inputs utilized to generate the base materials and masks as well as the used blending modes.

Motivation: Consider a project utilizing photogrammetry. The *layeredLit* shader, included in the *HDRenderPipeline* package, provides all the functionality needed. Different sources explain the workflow in detail (e.g., webblogs and articles like this one [58]) and facilitate the adaptation to the personal pipeline. The deadline to deliver the environment assets is in a few weeks. The texturing pipeline is highly texture based and already utilizes the *Substance Painter*. The desired visual quality and texel density cannot be achieved by pre-baked texture, *DP 03: Baked Texture Maps* (section 7.2.1). *Allegorithmics* layered material shader for *Unity* and *UE4* provides tools and a predefined pipeline. This is an easy way to incorporate pattern layering into your workflow. Because of its texture based base material and masking workflow, it is easy to use and allows the shipping of the final product in time.

Applicability:

- The requirements for the shader are already met by pre-existing shaders (e.g., *Unitys LayeredLit* shader for photogrammetry data).
- There is no development budget to create and test custom shaders and pipeline tools.
- In comparison to using *custom shaders*, see *DP 06: Custom Shader* (section 7.3.1.2), it affords less technical knowledge.
- The project does not require complete control over every single aspect of the rendering process.
- You prefer spending time in creating assets and art to struggling with technical challenges and shaders.

Implementation: *Unity* ships with the *LayerdLit* and *LayeredLitTessellation* shaders. When using the *HDRenderPipeline*, a new material can easily be created using these shaders. They combine object specific textures with tilable base materials, *DP 02: Hybrid Pattern Layering* (section 7.1.2). Other layered material shaders can be downloaded from *Allegorithmic's Substance Share* Website.[4] These shaders are available for both *Unity* and *UE4*. These are purely pattern based—*DP 01: Pattern Layering* (section 7.1.1)—and rely entirely on texture based material inputs (see pattern *DP 20: Textures* in section 7.7.1.1). Further information on the *LayeredLit* and *Layered Material* shaders can be found in section 5.2.4.

Examples: Different examples for built-in shaders can be found in section 5.2.4. *Unity's Fontainebleau* [45] demo illustrates the power of the previously explained *LayeredLit* shader. Figure 7.8 shows an exterior scene using photogrammetry in combination with a built-it pattern layering shader. The scene was created to represent a real game level. It targets a frame rate of 30fps on a standard PlayStation 4. Using a preexisting solution like this one can save months of development and testing effort.

Consequences:

Visual Qualities:

- The possibilities are limited by the used shader.
- These shaders are normally designed for general purposes as they are made to account for different situations and projects.
- They provide only a limited control over the amount of layers, inputs used for the individual base materials and masks as well as the blending.
- Procedural inputs for manipulating masking, base material creation and blending are most likely not implemented into the shader (e.g., vertex color, world space normal and world position)

Performance:

- These shaders are already optimized and hopefully tested on multiple platforms.
- They don't provide any control for optimizing performance usage.

[4] The following website contains layered materials shaders for both *UE4* and *Unity*: https://share.allegorithmic.com/libraries?by_category_type_id=15.

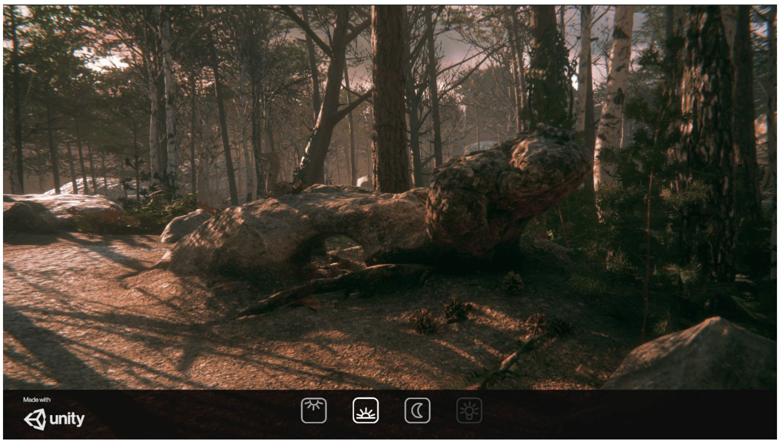


Figure 7.8: Fontainebleau [45], creating a realistic scene by combing photogrammetry with pattern layering. Image source: [59].

Pipeline (Workflow):

- Built-shaders handle complex blending of the individual parameters between base materials automatically.
- They are already integrated into predefined pipelines. These pipelines can easily adapted to the current project.
- They limit the possibilities layered materials offer dramatically.

Related Patterns: A Built-in shader is most probably implemented as a *DP 07: Uber Shader* (section 7.3.2.1) and does only use texture based inputs for the base material description and masking. This kind of shader is designed to fit many different purposes and to be intuitive.

7.3.1.2 DP 06: Custom Shader

Intent: Create custom shaders to control every aspect of the material layering process (e.g., material containers, masking containers and blending modules). This provides huge control over accuracy and performance as well as the used number of layers, used masking methods and utilized inputs. However, it comes with the cost of manually optimizing the performance, ensuring the proper blending of individual materials and testing all aspects.

Motivation: You are assigned with the task of shading and texturing a huge exterior scene. The scene consists of rocks surrounding a small lake. The moss is supposed to cover the top of the rocks. Areas close to the water should be covered by more moss than areas further away. Instead of creating individual masks for all the objects, you can utilize procedural methods. Using world space normals enables you to easily mask out areas facing to the top. Additional data, like world space position and tileable texture masks, can be used to add further detail. This procedural approach enables rotating the individual assets and all the masks adapt dynamically. Incorporating vertex color into the shader enables controlling the masking further. For instance, blue vertex color adds and red color removes areas from the mask. Custom shaders provide the flexibility to implement systems like this.

Applicability:

- Customs Shaders provide full flexibility in the amount and complexity of base materials.
- Using custom shaders is most likely the only possibility to use procedural methods for base material or mask definition within your material (e.g., masks based on world position normal or color tint depending on position).
- The masking between different base materials is not only based on texture masks but also incorporates different parametric inputs and scene data.
- A budget to develop custom layered material shader and additional pipeline tools exists.
- There is time to carefully test the layered material shader and its individual parts.
- The complete control over performance usage and optimization is needed and used.

Implementation: *Unity* and *UE4* provide different possibilities for implementing pattern layering. These methods are described in chapter 5. Both engines support the shading languague HLSL/Cg and node based shader graphs. Additionally, *UE4* provides two dedicated systems to create layered materials (see section 5.2.2 and section 5.2.3).

Examples Huge *AAA* video game productions like [34, 37, 41, 43, 44]use custom shading solutions. This provides the developers with the maximum control over visual appearance and performance. *Letzte Worte* uses custom shaders for most of the environment assets. Most of the patterns presented in this catalog were implemented and evaluated within this project.

Consequences:

Visual Qualities:

- Custom Shaders provide the most artistic freedom in terms of what is possible.
- A big part of the artistic process takes place while creating the shader.
- It represents a highly technical approach in the handling texturing.
- Shaders can be utilized to solve blending and material creation fully or partially.
- This approach requires a ot of technical knowledge.

Performance:

- Custom shaders provide the full control over resource usage and performance.
- Achieving good performance requires a lot of testing and optimization.
- Custom shaders are error prone for bad performance.

Pipeline (Workflow):

- Iterating on different base materials and masks is fast.
- Iterations on the implemented shader are difficult and slow.
- Creating stable shaders is time consuming and requires a lot of experience.
- A lot of testing is needed, especially if the shader offers many different switches and states.
- Shaders need to be tested on all applications the final product appears on. Consoles and mobile devices may not support certain features the shader uses.
- You have the power and responsability to take full controll over features, performance and usability.

Related Patterns: Patterns *DP 07: Uber Shader* (section 7.3.2.1), *DP 08: Individual Shader* (section 7.3.2.2), *DP 09: Content Generated Shader* (section 7.3.2.3) show different ways to include custom shaders into the project pipeline. Patterns in the categories *Material Container*, *Masking Container* and *Blending Module* provide further information on how to design and implement the individual components.

7.3.2 Workflow

Organizing and structuring your assets are important tasks. The following patterns discuss different methods on how to choose and design shaders and systems to fit into what your workflow needs. *DP 07: Uber Shader* (section 7.3.2.1) illustrates a workflow with few centralized multi-purpose shaders that cover most of the use cases within your project. Changes, fixes and optimizations within the shader are global and apply to all materials using this shader. Every additional feature increases the complexity of the shader. *DP 08: Individual Shader* (section 7.3.2.2) demonstrates the opposite approach to creating case specific shaders. Individual shaders do only incorporate the functionality for a unique purpose and are therefore more lightweight.

7.3.2.1 DP 07: Uber Shader

Intent: Create shaders to fit all or most of the needs for a specific kind of objects, like all environment assets or characters. This way, changes are propagated to all materials.

Also Known As: *Master Material*

Motivation: Consider a project with many different interior scenes. The modular, wall and floor assets are supposed to use pattern layering. The features required for the shader are clear: UV or world position based texture mapping, the base materials and masks are texture based. Additionally, it is possible to add a random tint. A technical artist designs the shader. The shader is tested by the artists creating a test scene and simultaneously optimized to work properly on all devices.

Applicability:

- When working with built-in shaders (*DP 05: Built-in Shader* in section 7.3.1.1), you are most likely using an *Uber Shader* as they are generally

designed to fit general purposes.
- The shader is supposed to work for a wide variety of situations and probably for different projects.
- A multi-purpose shader is created by a developer or technical artist, tested extensively on all devices the project is developed for. The shader is handed over to artists and not supposed to change dramatically.
- The feature requirements for the shader are limited. Most of the materials need similar parameters.

Implementation: The implementation of an uber shader is not different from implementing a individual shader, see *DP 08: Individual Shader* (section 7.3.2.2). The difference is mainly in the design, i.e., which parameters are exposed to the individual materials. Uber shaders and individual shaders can be used to complement one another in the same project.

Examples: Many of the environment assets within *Letzte Worte* use the same uber shader. Especially, the first location does almost exclusively use uber shaders to define all materials. Figure 7.10 shows one of those assets. They were initially used to unify the shading workflow and make it easy to change individual materials. In the later stage of the production, I created more individual shaders to reduce the texture count and to better incorporate procedural methods into the shader creation.

Consequences: Visual Qualities:

- The visual posibilites are provided by the uber shader. Every feature needs to be implemented in this shader. The visual variety and style is therefore highly influenced by it.

Performance:

- Extensive testing is necessary to ensure an efficient performance.
- It shares all performance limitations of pattern layering (see *DP 01: Pattern Layering* in section 7.1.1).

Pipeline (Workflow):

- Changes and modifications are propagated across all materials using this uber shader, i.e., fixes as well as errors are applied to all materials instantly.
- The shader can get really complex to account for all cases needed.
- Maintaining and sharing this shader with others can get complicated due to its complexity.
- The shader may offer a lot of flexibility for the artist in enabling and disabling shader features.
- Extensive testing is important because the shader has to account for many different scenarios and later changes will influences all materials using this shader.

Related Patterns: Built-in shaders, *DP 05: Built-in Shader* (section 7.3.1.1), are generally implemented as uber Shaders, due to their intent on fitting general purposes.

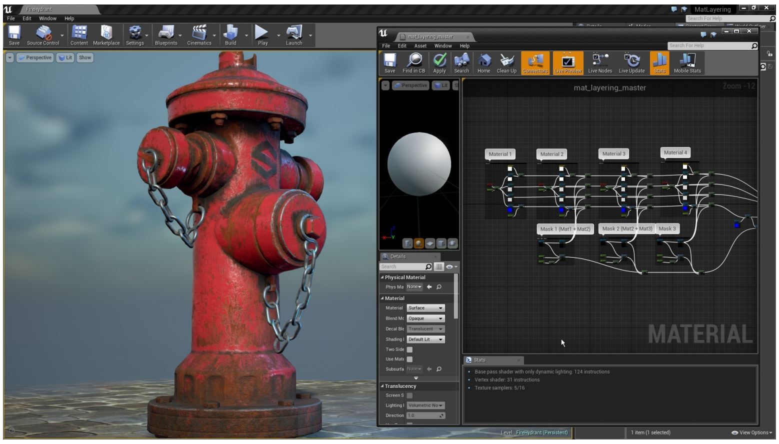


Figure 7.9: The material layering shader provided by *Allegorithmic* for *UE4*. Image source: [24].

Figure 7.10: An assets from *Letzte Worte* using a standardized *Uber Shader*.

Further, they generally use a full material properties description, see *DP 12: Full Material* (section 7.4.2.1).

7.3.2.2 DP 08: Individual Shader

Intent: Create individual shaders to fit the requirements of unique or problem specific needs. These are single purpose shaders and not intended to be used in many different ways.

Motivation: Consider a hero object within your scene. The player triggers a game play element by touching the object and so cause the object to change the material state. The object material transitions from a clean golden metal into an old, aged version covered with dirt and dust. This shading feature is unique to this particular case. In an other example, a few objects within your scene are made of fur. The shading requirements are really similar for all of them. You create a shader with custom fresnel, parallax occlusion mapping and other features specifically designed for this fur shading.

Applicability:

- The shader does only need to cover one specific use case. It is unique in its requirements. Incorporating this functionality into a uber shader would only increase the complexity of the shader.
- It enables a more artistically driven shader creation. For instance, specific shaders can be created to fit the artistic goal of an object.

Implementation: Individual shaders are implemented either as *DP 01: Pattern Layering* (section 7.1.1) or *DP 02: Hybrid Pattern Layering* (section 7.1.2) and use most likely *DP 06: Custom Shader* (section 7.3.1.2)

Examples: *Letzte Worte* uses individual shaders for specific use cases. Figure 7.11 shows two of them. The carpet uses a specific shading feature and therefore uses a special shader specifically designed to fit these requirements. The laptop is a gameplay element the player can interact with. This individual shader provides features like switching the screen on or off and changing the desktop of the screen.

Consequences:

Visual Qualities:

- An individual shaders offers lot of artistic freedom. The shader does only need to fit the demands of few use cases.
- It is much less important to create a user friendly and predictable shader than it is for an uber shader. Only parameters for this use cases need to be exposed, and most likely only a few people will ever work with this shader.
- The shader does not need to fit into the design of an uber shader and is therefore more flexible.

Performance:

- These shaders share all performance limitations of other pattern layering shaders. Testing is essential.
- As they are used only a few times within a project, performance is generally less important than for an uber shader that is used all over.

Pipeline (Workflow):

- Changes within the shader are only applied to the individual shader and the few materials using it.
- The shaders are generally less complex than uber shaders as they do not need to cover for all cases.
- The shader development is more artist driven.
- Aspects like performance and usability are not as important as for uber shaders.

Related Patterns: The only difference to an uber shader, *DP 07: Uber Shader* (section 7.3.2.1), is the different complexity and that their usage is limited to only a few use cases.

Figure 7.11: Using individual shaders for unique and specific requirements. Both the carpet and the laptop use individual shaders as they have specific shader requirements: the carpet uses parallax occlusion mapping for the hair while the laptop material changes states depending on the player interaction (e.g., switching on by opening the laptop).

7.3.2.3 DP 09: Content Generated Shader

Intent: Use a tool to create automated custom shaders from user given material layers, masks and parameters. The shader is generated automatically based on the inputs, instead of the desired inputs being specified by the shader. This turns the regular workflow—i.e., *DP 08: Individual Shader* (section 7.3.2.2) and *DP 07: Uber Shader* (section 7.3.2.1)—around.

Also Known As: *Material Masking System (MMS)*,*Material Layers* in *UE4*

Applicability:

- Writing your own tool will not be an option for most cases. Content generated shaders requires highly sophisticated tools. A huge amount of research, development and testing runs into creating such a tool.
- An appropriate tool exists that works with the given real-time engine. The tool is stable, has been tested and covers the needs for the given project.
- A shared, art directed material library does already exists or is planned to be built.
- A huge variety is needed for combining different base materials, masks and blending modes.

Implementation: Developing a custom tool is complicated as it has to create shader code automatically from arbitrary material, masking and blending inputs. Additionally, the inputs cannot simply be linearly interpolated but need to be blended differently depending on data types, parameters and shading algorithms (see section 4.4.1 for further details). To the best of my knowledge, *Unity* does not provide any system for this. *UE4* has just recently implemented a corresponding system with their *Material Layers* (see section 5.2.3 for more details).

Examples: *Gears of War 4* [44] uses a corresponding system for their material pipeline [19]. See figure 7.12. Another tools is provided by *UE4* as shown in figure 7.13.

Consequences:

Visual Qualities:

- The artistic freedom of contend generated shaders is dependent on the system used. Generally, they allow to specify an arbitrary number of base materials with utilizing different masking methods and blending modes in a user friendly way. Additionally the system automates complicated technical processes of properly blending.
- Individual shading features can be used for different assets.

Performance:

- Ideally the content generated shader does a lot of performance optimization automatically.
- The tool creates input driven individual shaders. Please refer to *DP 08: Individual Shader* (section 7.3.2.2) for further details.

Pipeline (Workflow):

- The tool provides an easy and fast way to combine and replace material layers.
- Material layers can be replaced by materials using a completely different structure.
- The system provides a modular way to define masking between individual layers.
- It is user friendly as the individual components are completely independent and can contain arbitrary data and computation.

Related Patterns: *DP 09: Content Generated Shader* is a system to create *DP 08: Individual Shaders* (section 7.3.2.2) based on the base material inputs, utilized masking method and blending mode.

7.4 Material Container

The following categories correspond to the material layering model defined in chapter 4. These components are: material container, masking container and blending module. Each category will contain patterns on how to plan and design these components. This first section focuses on the material containers. Core elements of every layered material shader or system are the individual base materials. These base materials are described in the material containers. They use different inputs, can do further computation within the module and output a material layer. These material layers define the surface type and appearance of the final object. The following sections deal with different methods of how to organize and split a surface into individual base materials. They provide patterns on how to define the complexity of individual material containers. Finally, they offer patterns on how and why to use distinctive inputs from within the material container to compute the final material layer.

Figure 7.12: A scene from *Gears of War 4*. It was created by using their custom material layering tool, the *Material Masking System*. Image source: [26].

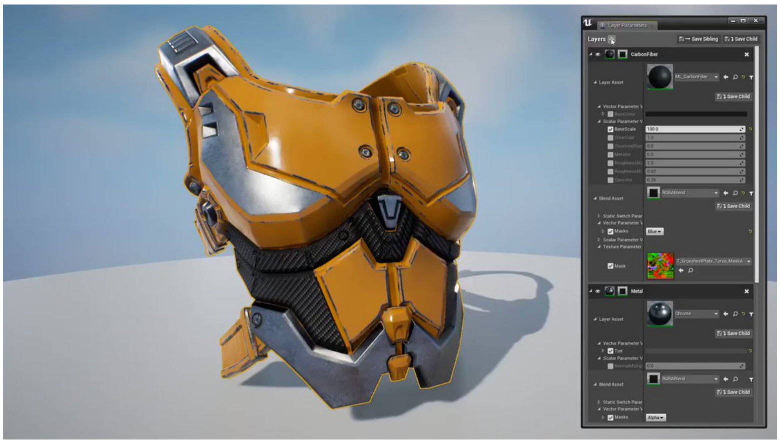


Figure 7.13: The *Material Layer* system by *UE4*. Image source: [33].

7.4.1 Granularity

As mentioned before, patterns *DP 10: Base Material* (section 7.4.1.1) and *DP 11: Material Variation* (section 7.4.1.2) describe different philosophies on how to split a surface into individual base materials. *DP 10: Base Material* proposes to split a surface by distinctive generic surface types (e.g., metal, rubber, wood, painted wood). Most of the complexity is created by blending those simpler base materials together. Pattern *DP 11: Material Variation* describes a method of using more complex base materials containing different surface types and creating different variations of them. This enables the creation of more complex surfaces with fewer base materials but also decreases the use cases for individual base materials.

7.4.1.1 DP 10: Base Material

Intent: Use individual generic base materials to emulate different surfaces and their properties, such as wood, painted wood, steel, rubber etc. The individual base materials should be generic enough to be reused across many different objects.

Motivation: You define a base material library within your project. Recurring base materials like different kinds of wood, metal and fabric get stored in it. Whenever you need a new material, you can add it to the library. The library grows over time. You can texture a new asset by reusing all these generic materials you have already created.

Applicability:

- Base materials are used to describe different surface types (e.g., wood, oak wood, painted wood).
- The base materials are limited in their complexity. They represent one specific surface type (such as rubber).

Implementation: If you use *DP 10: Base Material* or *DP 11: Material Variation* is implementation independent. It is a primary a decision of workflow and how to split a surface into base materials. These are rather workflow decisions that simply define how to handle individual layers.

Examples: *Letzte Worte* has an exterior scene. See figure 7.14. A path leads from the starting point to the top of the mountain. The environment is textured by reusing the same base materials. These base materials are mainly: rock, moss, ground and forest floor. Each base material represents a different surface type. They are generic enough to be used across different objects.

Consequences: Visual Qualities:

- It is easy to change and reuse base materials as they represent surface types.
- The number of different surface types is limited by the shader. Using many base materials increases performance costs and is therefore limited to a few. This decreases surface variety.
- Visual quality and diversity is highly controlled by blending, not necessarily by the base materials.
- The blending quality is highly dependent on masking resolution.

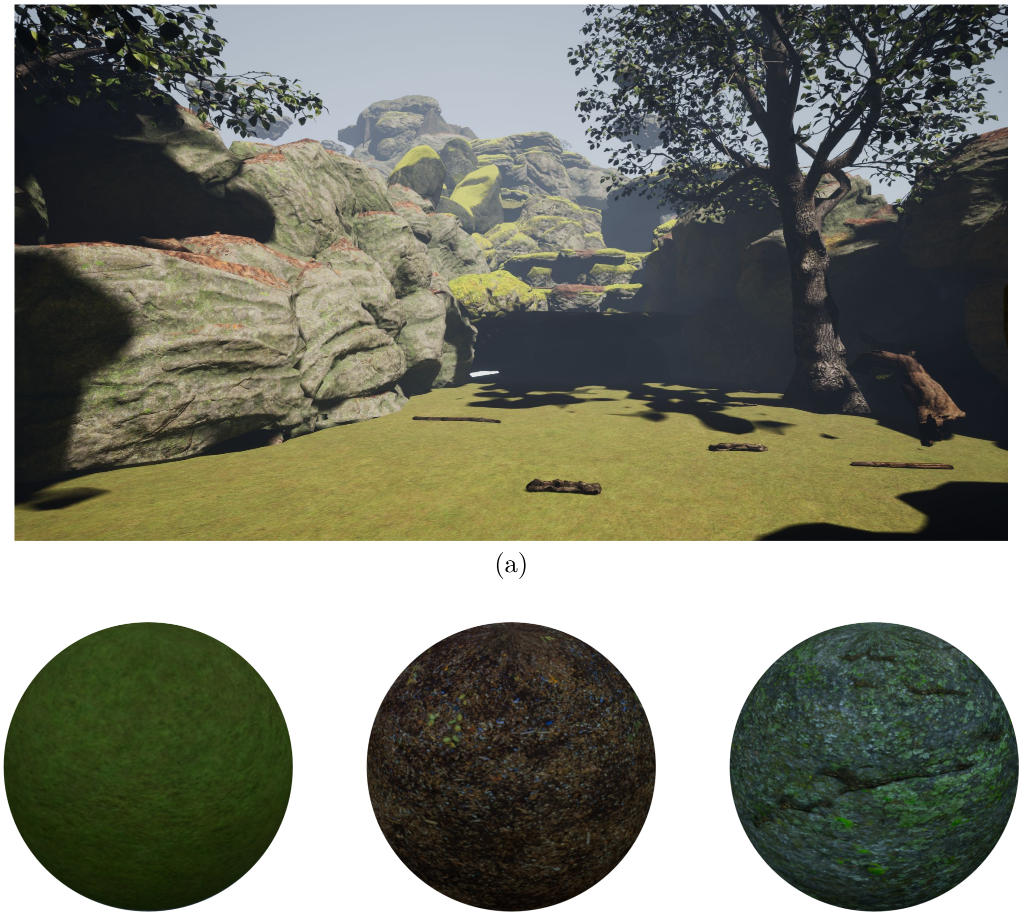
Performance:

- Using a high amount of base materials increases performance costs.

Pipeline (Workflow):

- The base materials themselves can be really simple.
- Creating different variations of the shading is simple and fast. You can simply swap two base materials.

Related Patterns: This pattern presents, additionally to *DP 11: Material Variation* (section 7.4.1.2), a proposal for how to split your surface into different base materials.



(a) (b) (c) (d)

Figure 7.14: An exterior scene from *Letzte Worte*. This scene uses mainly these base Materials: grass (b), forest ground(c) and rock (d).

7.4.1.2 DP 11: Material Variation

Intent: Rather than creating a set of different generic base materials representing distinctive surface types, create variations of more complex material.

Motivation: For instance, you want to create a kitchen scene. The walls are covered with ceramic tiles. Some areas that are often used should appear dirties and have small damages while other areas are supposed to look newer. To do so, you can create both a new and clean version as well as an aged, slightly damaged and dirtier one of the same base material. This enables you to have more surface variety within the materials than by combining generic base materials. Further, consider a cobblestone surface which could be recreated by combing stone, mud, wood chunks, dirt and moss as base materials. Instead of combing them within the engine, they are baked into textures and tiled across the surface area. Material layering is afterwards used to create additional variations of this material.

Applicability:

- The surface is extremely complex and composed of a huge variety of different surface types.
- The purpose of the material layering system is to create a variety of the surface types rather than blending completely different ones (i.e., it is used to add aging and damage to certain areas).
- The masking does not need to be as precise as for *DP 10: Base Material* (section 7.4.1.1). Therefore, the texture resolution can be reduced.

Implementation: This approach can easily be used with all pattern layering methods.

Examples: The first interior location in *Letzte Worte* (see 7.15) uses white wood material across all doors and windows. Initially, I split the surface into its distinctive surface types. The mask resolution was huge to achieve all the fine details, like for instance scratches. Instead of splitting the surface by the surface type, I later created two variations of the white wood material, a clean and a damaged one. This results in a highly decreased mask resolution, compared to before. The texture atlas uses only a texture resolution of 1024 by1024 pixel for the final scene, containing the mask for two doors and two windows.

Consequences:

Visual Qualities:

- This approach allows to have more distinctive surface types than using *DP 10: Base Material* (section 7.4.1.1).
- Different aging or damaging stages of a material can be defined and blendend afterwards.

Performance:

- This method requires less but more complex base materials in comparison to *DP 10: Base Material* (section 7.4.1.1).
- The masking resolution can probably be reduced highly as it is rather used for masking out areas instead of creating transition between distinctive surface types.

Pipeline (Workflow):

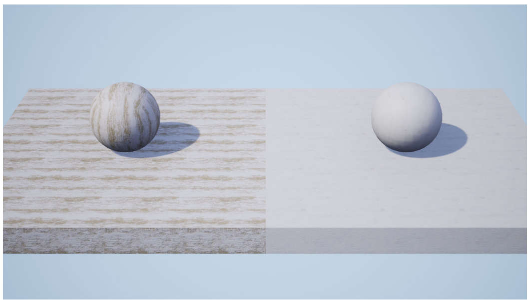
- Base materials are more specific and can be used across less different objects.
- This approach can be used with basically all pattern layering systems.
- It does most likely built on texture data input.

Related Patterns: In addition to *DP 10: Base Material* (section 7.4.1.1), this pattern presents an option in how to split your surface into different base materials.

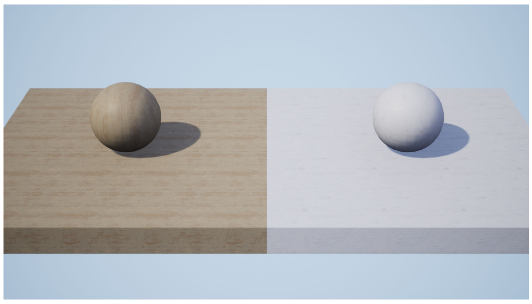
7.4.2 Complexity

The patterns proposed in this category help you to define how much complexity and data you want to put into a material container. Does every material container represent a fully described material layer object defining all surface properties? Can material containers be used to describe only a few material properties? *DP 12: Full Material*

(a)



(b)



(c)

Figure 7.15: Splitting a complex surface into either distinctive surface types or material variations. Figure (a) shows the final scene from *Letzte Worte*. The doors are made by blending material variations as seen in figure (b) rather than by combining two completely different surface types as seen in figure (c).

(section 7.4.2.1) and *DP 13: Modulation Layer* (section 7.4.2.2) compare these two distinctive workflows and will help you decide where to use which one. These patterns are closely connected to the patterns in category *Blending Module*: *DP 18: Physical Material Blend* (section 7.6.1)and *DP 19: Custom Material Blend* (section 7.6.2). The design of the *Blending Module* needs to provide the functionality for either blending only certain material properties or not. Section 4.4.2 contains further details how this decision is closely connected to the blending module.

7.4.2.1 DP 12: Full Material

Intent: Create a base material by specifying all material parameters (e.g., base color, roughness, metalness) so that it fully represents a real world surface.

Motivation: You use a physically plausible blending workflow as presented in pattern *DP 18: Physical Material Blend* (section 7.6.1). You specify base materials that represent real word surfaces and blend those.

Applicability: The intent is to work in a physically plausible way. Real world surfaces are composed of different surface types. Creating base materials that describe the entire material properties of a certain surface type corresponds best to this idea.

Implementation: This approach can be used with all pattern layering systems.

Consequences: Visual Qualities:

- The number of different base materials is highly limited as every one contains full surface description data.
- The results of blending two base materials with one another are predictable and therefore artist friendly.
- The workflow is intuitive if you are used to decomposing an object into different base materials (e.g., a metal, partially covered with rust, paint and dirt)
- The artistic process is much freer than *DP 13: Modulation Layer* (section 7.4.2.2) as blending behaves consistently. Besides, you don't have to worry about how to blend which individual material parameter.

Performance:

- Blending many similar base materials may result in blending similar roughness or normal values. By providing more control, this texture count could be reduced without a significant impact on the visual quality.

Pipeline (Workflow):

- The blending between different base materials behaves in a consistent way.
- The structure of all base materials is consistent. This makes it easy to change and replace them.

.

7.4.2.2 DP 13: Modulation Layer

Intent: Enabling material containers to specify only a few material properties instead of all makes them more lightweight but less predictive.

Motivation: The amount for blending fully described base materials is limited to only a few. To increase the number of base materials that can be blended a more flexible way of defining material containers is used. Instead of blending four wood base materials, all fully described with all parameters, you can blend two fully described base materials with partially described ones. Latter are used to modulate the roughness and replace the base color channel. This decreases the texture count from twelve (three per base material) to eight.

Applicability:

- You want more flexibility in if and how to blend the individual material parameters.
- The amount of layers can be decreased by reducing the complexity of individual base materials and the blending process.

Implementation: This approach requires the ability to influence the shader or shading system. It is therefore not viable for *DP 05: Built-in Shader* (section 7.3.1.1). It also requires more control over the actual blending process. With this approach, a material container does not represent an entire surface but only certain properties. It will therefore not work for pattern layering shaders or systems that target a physically plausible blending, *DP 18: Physical Material Blend* (section 7.6.1). For this pattern to work, a pattern layering system is required that uses the following patterns: *DP 06: Custom Shader* (section 7.3.1.2), *DP 19: Custom Material Blend* (section 7.3.1.2) and *DP 08: Individual Shader* (section 7.3.2.2) or *DP 09: Content Generated Shader* (section 7.3.2.3).

Examples: Figure 7.16 shows an interior location from *Letzte Worte*. The wall shader is composed of different layers: the main plastered wall, some rougher wall and three painted wall layers. These painted wall layers do not contain a full base material: they only include a base color and a value to adjust the roughness.

Consequences:

Visual Qualities:

- This approach allows to use more layers, especially if the distinctive base materials are supposed to be really similar.
- The considerations of how to blend which parameters, while considering the trade-offs between using certain material parameters or not, might hinder the artistic process.

Performance:

- Describing and blending base materials only partially makes them more lightweight.

Pipeline (Workflow):

- This introduces an additional stage within the pipeline and increases shading and pipeline complexity.
- The shader or shading system does require to fit certain defined criteria, further explained above in Implementation.

Related Patterns: For this pattern to work, the blending module requires to be designed according to pattern *DP 19: Custom Material Blend* (section 7.6.2).

7.4.3 Creation

The following patterns support you in defining the creation process of individual base materials. I decided to categorize them into *DP 14: Input Based* (section 7.4.3.1) and *DP 15: Semi Procedural* (section 7.4.3.2). In the former, the material properties are defined by the inputs without much additional computation (e.g., texture maps defining roughness, metalness and base color). The latter uses different inputs to drive procedural material creation (e.g., color tint based on world position). This decision has huge impact on the pipeline and way of working with base materials. These patterns can also be combined to achieve the desired result.

Figure 7.16: Blending partially described base materials to create surface variaty in *Letzte Worte*. The orange wall does not use a full base material description. It defines only individual properties to modulate the previous base material.

7.4.3.1 DP 14: Input Based

Intent: The surface properties of the base material are mostly defined by its inputs. There is limited to no manipulation of this data before passing it on to the shader or next component within the material layering module.

Motivation: You define the surface properties of the base materials within any 3rd party painting tool. After exporting, importing and assigning them to the right materials, you just want it to look the same as in the authoring tool.

Applicability:

- This works best in combination with other texture authoring applications.
- Values are set by either using textures or parameters.
- The result is predictable, highly controllable and well supported.

Implementation: An input defined base material creation is supported by any pattern layering implementation or workflow. Shaders using *DP 05: Built-in Shader* (section 7.3.1.1) require most likely input defined base materials.

Examples: The interior room from *Letzte Worte* in figure 7.17 uses mostly texture inputs for the definition of the base materials. Only a few objects use procedural shading, e.g., the color bottles, tubes and folders use procedural color randomization.

Consequences:

Visual Qualities:

- This provides a control over the individual properties of a base material.
- Artists can focus on creating art and do not need to get technical.

Performance:

- It is efficient if textures are reused across a lot of different objects.

Pipeline (Workflow):

Figure 7.17: A location from *Letzte Worte* using mainly input defined base materials.

- It uses an artist friendly workflow.
- Texture based workflows are widely integrated in the different pipelines.

Related Patterns: Input driven base material definition is commonly used across all pattern layering worfklows and implementations. A *DP 05: Built-in Shader* (section 7.3.1.1) is most likely limited to this input based workflow.

7.4.3.2 DP 15: Semi Procedural

Intent: Use input driven procedural methods to define and modulate the final base material properties.

Motivation: Consider a modular level kit which already uses pattern layering for shading. The same wooden wall panel asset is located several times side by side. It is noticeable that they share the same textures and UVs. An easy way to fix this is to procedurally offset the UV coordinates for the wooden base material. All masking and all other base materials stay in the same place. By offsetting the UVs on every object differently based on the shader, it is much less obvious that they share the same texture.

Applicability:

- Pseudo randomness is wanted from within the shader.
- Material properties need to interact with scene data (e.g., change of color based on scene height).
- The shader uses shader based vertex animation.
- The shader is supposed to interact with the player or other dynamic objects (e.g., interactive grass, white foam around objects in the water).

Implementation: The shader graph editors within *Unity* and *UE4* provide access to a huge variety of inputs and different operations for manipulating them.

Examples: The color tubes and bottles shown in figure 7.18 use the same textures. A semi procedural base material, combined with a mask, is used to randomize the color of the bottles. Additional color splatters are placed on top. The UVs for the bottles are randomized based on their world position so that each is covered differently with color splatters. This allows to generate an infinite number of distinctive color bottles by still using the same inputs for all of them.

Consequences:

Visual Qualities:

- Procedural systems can be used to increase variety.
- They can be used to create a more reactive and immersive world (e.g., reacting vegetation).
- They can create global changes across different objects.

Performance:

- The performance cost depends highly on the operations and inputs used.
- UV manipulations and simple mathematical operations are cheap.
- Recreating complex surfaces procedurally is expensive. Generally, only semi procedural methods are therefore used for creating photorealistic surfaces for real-time productions.

Pipeline (Workflow):

- They are easy to use with shader graph editors.
- Simple procedural input manipulation can be recreated manually in most content creation application like *Maya* and *Blender* for previewing.

Related Patterns: Using semi procedural base material definition can be used to create huge variety, in contrast to *DP 14: Input Based* (section 7.4.3.1). The inputs used are an important aspect for procedural creation. Please refer to category *External Inputs* in section 7.7. The operations used are similar to those for procedural masking *DP 17: Procedural* (section 7.5.2).

7.5 Masking Container

The masking container is responsible for generating the mask controlling influence and area of the blending module. A mask can be generated in many different ways by either using textures or procedural methods. Masking is a core element of any material layering system. The following patterns, *DP 16: Texture Based* (section 7.5.1) and *DP 17: Procedural* (section 7.5.2), will help you planing and setting up your masks.

7.5.1 DP 16: Texture Based

Intent: Use textures to mask out areas.

Motivation: You are working on a wooden floor. You want to blend different variations of the floor (e.g., new and clean, sun bleached, dirty and witch scratches) by using texture masks. Within the painting application, you paint sunlight affected areas red, areas that are supposed to have more scratches green and dirty areas in the

Figure 7.18: Procedurally modified base materials. All these color bottles share the same texture information (e.g., base color, normal, roughness). An additional mask channel is used to define the area that is procedurally modified to use a pseudo random color based on the position.

color blue. You import this mask into your game engine and use the individual channels directly as mask for the blending of the base materials.

Applicability:

- The base materials are blended by using texture masks.
- A textures is either used to create high detail blends or mask out larger areas.
- The blending is uniform and can be described by a scalar value.
- The masking is supposed to be shared across all objects using the same material.

Implementation: These inputs simply get exposed by the shader and set within the editor in the materials panel. The kinds of available inputs are defined by the shader. Using masks for blending within the shader graph editor is straightforward. Different masks are often combined in a single texture by using different color channels.

Examples: Most projects use texture based material masking. Figure 7.19 shows an interesting example from *Paragon* as this uses two different texture masks. The former defines bigger areas covered by a distinctive surface type. The second mask is used for additional surface variation, like scratches and dirt.

Consequences:

Visual Qualities:

- It provides high control over the blending process on a per pixel basis.
- Using texture masks provides an artist friendly, consistent and predictable way of using masks.

Performance:

- The impact on performance is highly dependent on number and resolution of used texture masks. Please refer to pattern *DP 20: Textures*

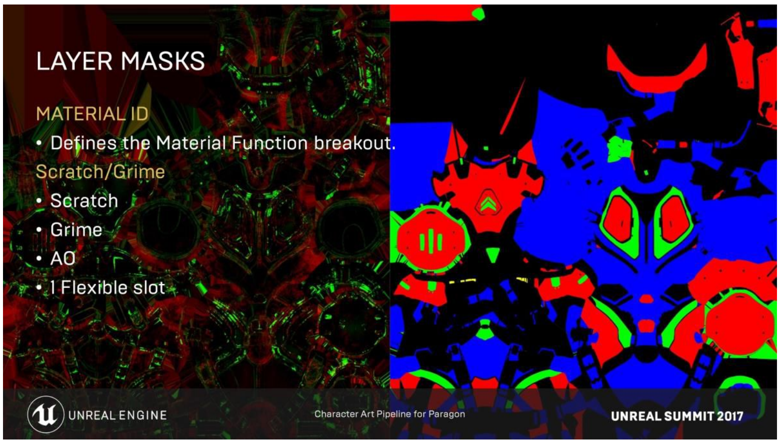


Figure 7.19: ID maps used as layer masks for a *Paragon* character. The right one defines the distinctive base materials like rubber, metal and plastic. The left contains the ambient occlusion map with additional detail masks for scratches and grime. Image source: [16, p. 116].

(section 7.7.1.1) for further information.

Pipeline (Workflow):

- Textures are easily integrated into the pipeline and shaders.
- Generally, working with texture masks is artist friendly as they are used to work with textures.
- Results can easily be shared across different applications.
- There are many different possibilities to create, generate and paint texture masks.

Related Patterns: More detail on why and how to use parameter inputs will be given in pattern *DP 20: Textures* (section 7.7.1.1). *DP 07: Uber Shader* (section 7.3.2.1) and *DP 05: Built-in Shader* (section 7.3.1.1) do most likely incorporate only textures based masks as they are designed to fit universal requirements.

7.5.2 DP 17: Procedural

Intent: Use procedural methods for the dynamic and automated generation of masks.

Motivation: You are presented a scenario of using procedural masking for exterior environment scenes in pattern *DP 06: Custom Shader* (section 7.3.1.2). In that example, procedural masking is used to generate moss on top of rocks. This can be achieved by utilizing the mesh independent world space normal vector. A procedural setup that is independent from object specific data can be reused across arbitrary objects without any additional manual work. A new rock assets can be added to the scene. By assigning the proper material, it automatically uses the proper masking. Further, additional features like vertex color can be implemented to control the blending process further. Consider working on an exterior scene and in the middle of the map is a small hill. To create a smooth transition from forest ground to greenfield, you incorporate a height based masking which blends automatically between both ground base materials.

Applicability:

- Procedural approaches can be used to create a system solving recurring problems. Instead of painting the moss on top of all rocks, you can automate the moss masking based on the world space normal.
- A requirement for the corresponding mask is to react to different inputs (e.g., height based base material blending).
- A dynamic masking system that can change at runtime (e.g., puddles reacting to a global weathering system, dynamic changes between material states, flickering emissive materials).

Implementation: Using and controlling procedural masks requires custom shaders, *DP 06: Custom Shader* (section 7.3.1.2). *UE4* and *Unity* provide with their shader graph editors powerful tools. They enable you to easily access different input data and utilize it to drive the procedural masking.

Examples: Figure 7.20 illustrates a layered rock shader using a procedural mask for the moss. The moss covers automatically the top of the rock. This even works by rotating the object within the 3D scene. It also works for different game objects using the same mesh and material with different rotations. The moss will always cover the top for all of them.

Consequences:

Visual Qualities:

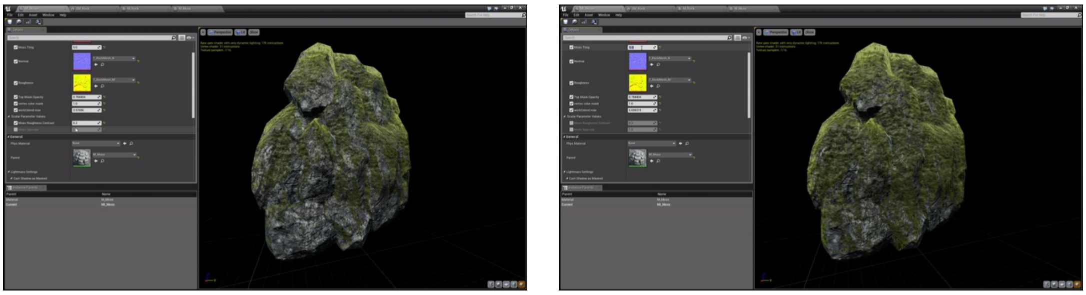
- A lot of visual work is done within the shader.
- Working with huge amount of different parameter inputs to control the masking might be unintuitiv.
- If and to what extent the artist is able to control the masking depends on the shader. Does it enable the artist to control or influence the masking process (e.g., by utilizing vertex paint within the engine)?
- This approach requires a rather abstract, logical and technical way of thinking about artistic problems.

Performance:

- These masking instructions are calculated at runtime. The performance depends highly on the amount and complexity of these procedural systems.
- As they are less reliant on stored data, they generally use less memory than textures.

Pipeline (Workflow):

- The shading pipeline has to support custom shaders.
- Masks can be reused across different assets and projects.
- It includes all the advantages and disadvantages of a procedural workflow.



(a) (b)

Figure 7.20: Procedural masking of moss. A procedural shader is used to create the layer mask dynamically. The moss is blended automatically to cover the top of the rocks. Different parameters can be used to control the masking amount, direction an sharpness: less (a) and more (b) moss.

Related Patterns: To use flexible procedural masking, custom shader solutions are required as explained in pattern *DP 06: Custom Shader* (section 7.3.1.2). This pattern is often used in addition with pattern *DP 08: Individual Shader* (section 7.3.2.2). Using individual shaders enables to create problem specific procedural masks. An easy way to include procedural masking is by using pre-existing content generated shaders as in *DP 09: Content Generated Shader* (section 7.3.2.3). Procedural processes are to a large extent defined by the inputs. Patterns within the category *External Inputs*, section 7.7, will be helpful in designing procedural masks.

7.6 Blending Module

The blending module is the last component of the material layering model. It inputs the material layers from the material containers and blends them based on a mask. Please refer to 4.4.2 for further information on this module. The following patterns, *DP 18: Physical Material Blend* (section 7.6.1) and *DP 19: Custom Material Blend* (section 7.6.2), illustrate two possible approaches for implementing such a blending module. The pattern *DP 18: Physical Material Blend* (section 7.6.1) represents a rather rigid system. It provides only a few blending modes. These blending modes try to be physically plausible and provide blending operations that respect the energy conservation law (except for adding emissive base materials). These blending modes define the mixing of all material parameters. Therefore, they are predictable, user friendly but limiting. The pattern *DP 19: Custom Material Blend* (section 7.6.2), on the contrary, does not try to stay physically plausible but enables the user to blend the different parameters of the material layers individually. The user defines exactly which material parameters are skipped, blended and how they are blended. This enables the use of material layers that describe the surface only partially, see pattern *DP 13: Modulation Layer* (section 7.4.2.2). This allows more flexibility in how to use the resources and thus makes the use of additional layers possible. The advantage as well as disadvantage for the user is the higher control and responsibility to weigh up performance versus accuracy and artistic freedom.

7.6.1 DP 18: Physical Material Blend

Intent: It uses a closed an rigid blending module that is limited to only a few modes. These blending modes are designed to be physically plausible and and user friendly, producing predictable and consistent results.

Motivation: Probably the easiest way to achieve realistic looking results is by imitating real world behavior. Constraining the blending modes makes it easier to ensure a proper blending (e.g., maintain the energy conservation law).

Applicability:

- This approach creates predictable and consistent results.
- Inputs are most probably mainly texture based.
- The goal is to use a physically plausible system.

Implementation: This design of the blending module can be used for all implementations. Almost any general purpose shader, see pattern *DP 07: Uber Shader* (section 7.3.2.1), uses a blending module based on this design as it is flexible enough to represent all kinds of base materials and does provide a good usability. It works best with base materials using *DP 12: Full Material* (section 7.4.2.1). Other shaders using pattern *DP 13: Modulation Layer* (section 7.4.2.2) do not work well for this concept.

Examples: Figure 7.21 shows an asset from *Letzte Worte*. The material is composed of different fully described base materials. These base materials are reused across different assets in this location.

Consequences:

Visual Qualities:

- Using this physically plausable blending approach creates predictable and consistent results. Therefore, it is easy to work with.
- The artistic process can focus mainly on creating appropriate base materials and the corresponding masks.
- It enables the artist to test different ideas and concepts out as materials can easily be changed.

Performance:

- Every base material uses a full material properties description (see pattern *DP 12: Full Material* (section 7.4.2.1))
- The performance cost is predictable as all the different options are limited.

Pipeline (Workflow):

- The workflow is straight forward. The energy can be spent on creating base materials and masks.
- Base material complexity and definition stay consistent throughout the project.

Figure 7.21: An asset blending fully described base materials. This image shows an early asset from *Letzte Worte*. All base materials use a full material description (e.g., base color, metalness, roughness, ambient occlusion): wood, dirt, dust, fabric, used fabric, fine plastered wall and rough plastered wall.

Related Patterns: The blending mode is highly connected with the complexity of the base materials. This approach will not work for base materials only defined partially, see *DP 13: Modulation Layer* (section 7.4.2.2).

7.6.2 DP 19: Custom Material Blend

Intent: This blending module design is much more open and provides the possibility to define exactly which, if and how individual material parameters are blended. It does not follow the constraints of trying to be physically plausible. This flexibility allows to define more precisely how resources are used and where to sacrifice accuracy for performance.

Motivation: Consider working on a complex layered material with a huge amount of different surface types. Consider a brick wall that is partially plastered. Both base materials use the pattern *DP 12: Full Material* (section 7.4.2.1) and are described in all their properties. Additional layers are introduced to add variety. For the plastered wall they are as follows: a water damaged variation, ground dirt, a newly plastered part and a slightly damaged variety. For the brick the following variations are added: a color variation and a water damaged version. Considering three textures per base material results in eight base materials using twenty-four textures. If we use only the base color texture for each variation and simply subtract or add a small scalar to the roughness value depending on the variation, the texture count gets divided by half. The result might not be physically accurate but it pays off as far as performance increase: a lower amount of used texture memory and computation instructions.

Applicability:

- This approach is well suited for large amounts of similar base materials.
- It works well if only a single material property is replaced.
- Custom material blending works great creating surface variation, e.g., puddles and water. Wetness can easily be achieved by manipulating the normal map, darkening the base color and decreasing the roughness.

Implementation: To achieve this high control over how to blend which individual material parameter, it is necessary to use custom shading solutions, *DP 06: Custom Shader* (section 7.3.1.2). Generally, this will require *DP 08: Individual Shader* (section 7.3.2.2) as this provides most flexibility. Using *DP 09: Content Generated Shader* (section 7.3.2.3) is the easiest way to incorporate this aspect into your pipeline.

Examples: Figure 7.22 shows a small example for this design pattern. Using pattern *DP 18: Physical Material Blend* (section 7.6.1) would result in either the need to create a duplicate base material representing the floor wood surface covered with water or need to to create an arbitrary blend mode that can combine an arbitrary transparent base material stacked on top of another. The former is unflexible as a wet material variation for any base materials needs to be created. The latter is technically challenging and has probably a huge impact on performance (see section 3.2.4 in chapter 3).

Consequences:

Visual Qualities:

- It enables the combination of a great number of different base materials.
- Blending is based on visual appearance rather than any physical plausibility.

Performance:

- Reducing base material and blending complexity affects performance positively.

Pipeline (Workflow):

- The pipeline needs to provide more control over if and how individual material parameters get blended.
- The best way to incorporate this pattern into your project is by using *DP 09: Content Generated Shader* (section 7.3.2.3).

Related Patterns: This approach works, in contrast to the previous *DP 18: Physical Material Blend* (section 7.6.1), with all base materials well, both using *DP 12: Full Material* (section 7.4.2.1) and *DP 13: Modulation Layer* (section 7.4.2.2).

7.7 External Inputs

This category presents different patterns associated with utilizing different external inputs. In this case, external refers to data that is not embedded within the shader, like

Figure 7.22: The water material layer is not described by all material properties. The appearance of water on top of a surface is created by using custom blending of individual properties.

parameters, mesh data, object data and scene data. These different inputs with their distinctive advantanges and disadvantages will be discussed in the following categories.

7.7.1 Parameters

This category includes basically all parameters that can be exposed by the shader to be set in the material (e.g., textures, vector and scalar paremeters). The following patterns present different use cases for this kind of inputs. As textures play such a fundamental role in shading, the design pattern *DP 20: Textures* (section 7.7.1.1) focuses entirely on them. *DP 21: Variables* (section 7.7.1.2) present other types of parameter inputs, like colors, position vectors, scalar value and switches. In pattern *DP 22: Scripted Parameters* (section 7.7.1.3), I want to illustrate possibilities to utilize scripted parameters.

7.7.1.1 DP 20: Textures

Intent: Use textures as a data table to define different material properties, masks or as input for procedural operations.

Motivation: Consider a project that targets a photorealistic art style. Textures are ideal as they are perfect to store complex surface information of all kinds. Furthermore, a wide variety of processes to generate textures exist (e.g., 3D scanning, photo editing, baking of procedural or high detailed 3D data and painting). Specialized texture tools make it quite easy to generate realistic looking textures.

Applicability:

- Complex surface information can be stored within textures easily.
- Huge libraries of high quality textures exist and can be utilized for the current project.
- Textures can be baked down from procedural approaches, vertex color or different 3D data.
- Textures can be used to store arbitrary data.
- A lot of applications are especially designed to generate, create and paint textures.

- Artists are used to working with textures. It is a familiar workflow.
- Commercial engines like *UE4* and *Unity* have already been well optimized for working with a large number of bigger textures. They use technologies like mid-mapping, texture streaming and advanced compressions.
- Textures are set per material. All assets using the same material share the same textures.

Implementation: A wide variety of different tools and approaches exist to create and generate textures. Textures can be created of photos, 3D scans or by using procedural methods.

Consequences:

Visual Qualities:

- Complex surface data can easily be created and stored.
- Textures generally provide the most efficient way to store huge surface data.
- Artists are used to working with textures. A lot of tools make working with textures intuitive.
- A lot of tools exist to generate, create and paint textures.
- Textures are projected onto objects. This projection can be manipulated to rotate, offset and scale textures. Textures can even be animated by manipulating these projection coordinates.

Performance:

- Textures have already been highly optimized through technologies like texture streaming, compression, mid-maping, filtering.
- Huge textures need a large amount of memory.
- The loading of large textures into the VRAM may take some time.
- The streaming of texture data has a huge impact on loading times.
- Methods like combining different objects into texture atlases and channel packing reduce the amount of textures.

Pipeline (Workflow):

- Textures can easily be created in other applications.
- Texture data is easy to share between different softwares.
- They can be stored in libraries and easily be modified and reused.

Related Patterns: Textures are used across all different pattern layering implementations and workflows. They are used within material containers to define base material properties directly or as an additional input to drive and manipulate procedural methods. They play an vital role in creating mask.

7.7.1.2 DP 21: Variables

Intent: Variables can be set per material or per component on advanced material layering systems. The most common parameters are textures, vectors, scalar values, booleans and switches.

Motivation: You want to specify different colors for different materials using the same shader. To do so, expose a color vector within the shader. This parameter can later be specified in the materials panel of the individual material. In another example, a procedural masking container uses the world position normals to mask areas facing the top. To add more control, you can expose an additional vector that enables you to influence the direction to be masked. In one material you want to influence the opacity of an individual material layer. Therefore, you add a scalar parameter to the shader and multiply it with the blend opacity of the material layer. Within the material you can now influence the material layer blend opacity.

Applicability:

- You wish to influence colors, vectors, opacities, switches and features on a per material basis.
- You expose parameters to control procedural processes.
- You want to tweak values in engine (e.g., adjust the roughness value slightly).

Implementation: In a *DP 06: Custom Shader* (section 7.3.1.2) you can simply define which parameters should be exposed to be set in the individual materials. In a *DP 05: Built-in Shader* (section 7.3.1.1) all available parameters are defined by the shader and can not be changed.

Related Patterns: Custom parameters can easily be exposed within a *DP 06: Custom Shader* (section 7.3.1.2)

7.7.1.3 DP 22: Scripted Parameters

Intent: Scripted parameters are regular material parameters accessed from within the game code to either set or change them at runtime.

Motivation: Imagine a flickering light bulb. You could either create the flickering by using procedural noises within the shader or by controlling certain parameters, like emissive color or intensity, from a script.

Applicability:

- Surface properties are supposed to change at runtime.
- The game logic should influence certain material properties.
- You want to access additional information that cannot be accessed from within the shader graph editor. Therefore, you want the game logic to set a property within the material, e.g., you can pass on the position vector of an arbitrary object within your scene to the shader using a vector parameter.

Implementation: *UE4* provides an artist friendly visual coding environment, the *Blueprints Visual Scripting* system. You can utilize this scripting environment to set and manipulate parameters within your materials.

Related Patterns: This pattern uses regular parameter values *DP 21: Variables* (section 7.7.1.2) that are set or manipulated at runtime by the game code.

Figure 7.23: A Layered Material from *Letzte Worte* using scripted parameters. This shader reacts to the player action. When triggered, the dark area grows to cover a big portion of the wall. The parameters to control this expansion are set and animated from the game code.

7.7.2 Mesh Data

The results of the previously mentioned input methods are applied globally to all meshes using the same material. Changes in the input parameters, like changing an id map for example, will inevitable effect all objects using this material instance. The inputs within this category are only influenced by the mesh data (e.g., UV coordinates, object-space normals and object space position). These inputs will be constant across all objects using the same mesh. *DP 23: UV Coordinates* (section 7.7.2.1) and *DP 24: Vertex Color* (section 7.7.2.2) show different methods to utilize these mesh specific properties within the shading process. This kind of parameter can be used to create variety across objects using the same material.

7.7.2.1 DP 23: UV Coordinates

Intent: Use the UV coordinates to influence, manipulate and animate the projection of textures onto the object.

Motivation: You want to add ornamental panels and trims to your scene. The intent is to combine them with pattern layering. The ornamental data for the panels and trims is sculpted within *zBrush* and finally baked down to a texture. The first UV set is set out nicely and does not have any overlapping areas. All UV islands stay within the UV boundaries. This first UV set is used for the tileable base materials as well as the object specific textures. The second one is arranged more freely to project the ornamental parts onto different areas of the mesh. This second UV map does not need to be set out as cleanly as the first (e.g., inconsistent texel density, overlapping areas and UV islands exceeding the UV boundaries). It is neither used to texture the entire object nor to generate the lightmap. You can duplicate this object and use the second UV map to project different details onto the copy. This allows to create variations by using the same materials in the engine. The mapping is changed by the UV set. As the first UV map is the same, object specific texture data can still be shared across these objects.

Applicability:

- It is used for adding detail that is independent from the first UV set.
- The UV coordinates can be used to offset textures, for example base materials to avoid repeating textures on the same object. This can be done by either offsetting UV coordinates pseudo randomly from within the shader or by exporting different meshes from the 3D content creation application.
- It can be used to blend different texture variations. Each variation is placed on a grid; offsetting the UVs by this grid will switch the material properties projected onto the object.

Implementation: 3D content creation software provides the functionality to add additional UV sets. These UV sets can be modified independently and sent to the game engine. It is important to ensure that the proper UV set is used to generate the lightmap, otherwise ugly artifacts may appear.

Examples: Trim sheets are an excellent example on how the UV coordinates can be used to influence shading. Figure 7.24 shows an example from *Gears of War 4*.

Consequences:

Visual Qualities:

- It provides high control where to put additional detail.
- This method allows to create highly detailed areas which can be sculpted and be reused across different assets sharing the same style.

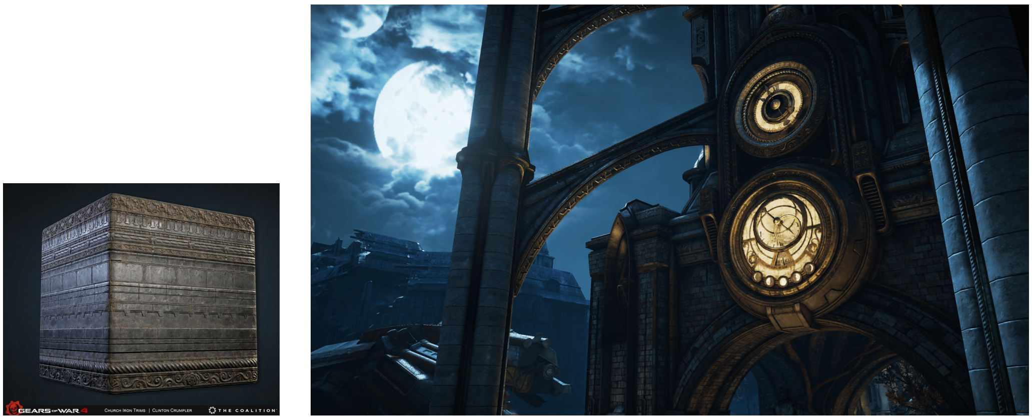
Performance:

- Creating variations by offsetting UV coordinates is easy and cheap.
- Animations done by manipulating the UV coordinates do not require sprite sheets where every single frame is saved as a individual picture.
- Using multi UV sets in connection with normal maps may cause problems. In *Letzte Worte*, normal maps caused problems when using the second UV set. The normal map effect appeared inverted for certain areas on the objects.

Pipeline (Workflow):

- Working with multiple UV maps can be challenging.
- Some software does not support multiple UV sets (e.g., *Substance Painter*).
- Engine shaders need to be re-built within the 3D content creation application to preview a similar result to the final in engine render.

Related Patterns: Different UV coordinates can be used in any *DP 06: Custom Shader*. Especially with procedural methods, it provides a powerful tool to create performance efficient variations, see *DP 15: Semi Procedural* (section 7.4.3.2) and *DP 17: Procedural* (section 7.5.2).



(a) (b)

Figure 7.24: A scene textured by combining pattern layering with trim sheets. Figure (b) shows a location from *Gears of War 4* combining pattern layering with trim sheets, see figure (a). Image source: [26].

7.7.2.2 DP 24: Vertex Color

Intent: Use vertex attributes to create per object variety independent from the material.

Motivation: Consider an often recurring object that might even be located side by side. This could be a fence, a book, a wall, a tree. Visible repeating patterns between the objects are noticeable because they use the same texture maps as a mask input for the material layering. Creating individual bitmap masks or materials is not really an option because of performance issues regarding draw calls and memory limitations. This problem can be solved by using vertex attributes. The vertex attributes are independent from the material and can be different for every object. This is also true if the objects share the same mesh or material. Each vertex has a vertex color attribute that stores RGBA values. These values can be retrieved as an input parameter within the shader and be used to manipulate the material parameters. The vertex color can be used directly to color the object or indirectly, for example as a mask. The value between each color channel can be used to mask out a material. Coloring and blending with vertex color is unique to a single object, in contrast to the id map method as mentioned before [50]. One huge constraint of vertex color is its limitation in resolution. The vertex paint resolution is directly connected to the vertex count of the mesh. To create sharp masks, a huge vertex count is necessary. This method alone is therefore mostly used to mask out bigger areas or in connection with other methods like height warp and brush maps.

Applicability:

- The used base materials do not change between the instances.
- The material blending takes place at a larger scale and not in detail space.

Implementation: The vertex color can be used as a mask to blend the different textures. This method is really powerful. The vertex shader normally implements four channels, rgba, which enables the artists to blend between five base materials.

Examples: Figure 7.25 from the *Unity* project *Fontainebleau* [45] shows how vertex color can be used to control the masking in pattern layering system.

Consequences:

Visual Qualities:

- Vertex colors can be painted within the engine.
- Generally, you can paint immediately adopting the final shader.
- The painting resolution depends on the geometry. Higher vertex count results in a larger painting resolution.
- It works great for masking out areas but not for painting really detailed small scale areas.
- Additionally, other techniques can be used to create more detailed textures (e.g., height warp, brush maps).

Pipeline (Workflow):

- Some engines do already provide vertex painting tools built-in, e.g., *UE4*.
- The masking takes place in the engine with the final shader.
- Vertex color can either be imported from the 3D content creation application or be overwritten from within the engine. By overwriting it in engine, re-importing changed mesh data, may destroy the painted masks.

7.7.3 Object Data

While inputs connected to the mesh data are independent from other objects and their location within the 3D scene, object specific inputs are influenced by them. Object related values like hierarchy, position, rotation, scale are object data inputs.

7.7.3.1 DP 25: Vectors

Intent: Use different vectors to manipulate the material based on location, scale and rotation.

Motivation: These vectors can be used to manipulate the other material parameter inputs. They could be used to manipulate an object depending on its position, rotation or scale. One possible use case for this can be to add a pseudo random variation—like random tint—to objects, depending on there position in space. The possible use cases are infinite as these vectors can be used to manipulate any parameter from color, saturation, roughness to UV coordinates. This kind of use does not work with moving object because the values update dynamically at runtime. Changing the translation, rotation or scale would therefore directly affects the material appearance at runtime.

Applicability:

- The objects have different location, scale or rotation values to drive procedural functions within the shader.

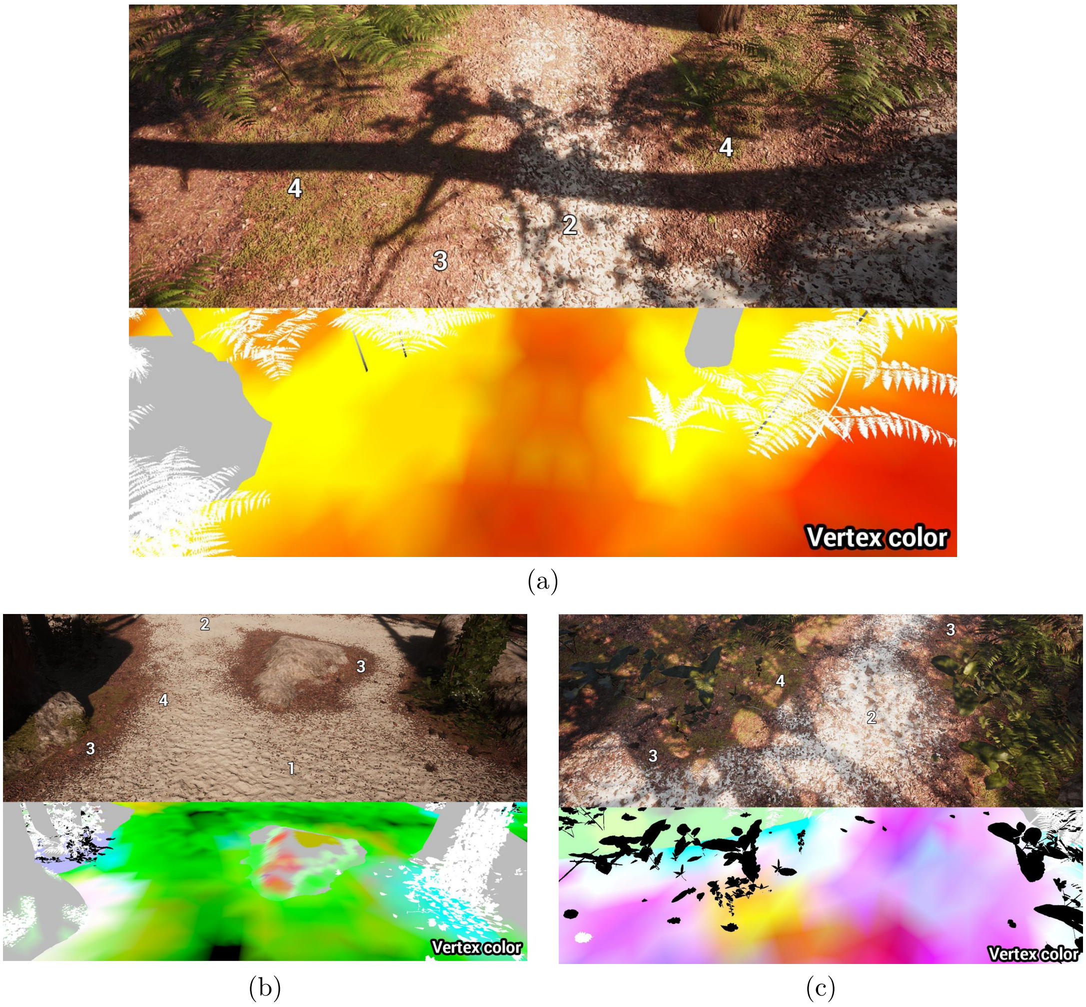


Figure 7.25: Using vertex paint to influence the base material blending and create surface variety. Image source: [58, p. 31–32].

- The specified vector does not change (e.g., If the random color is linked to the object position, the object should not be movable).
- Objects cannot be merged because it would combine them into one single object, with shared location, rotation and scale vectors. Therefore, all variety would be lost.

Implementation: *Unreal Engine 4* allows to access the object position, orientation and bounding box data really easily within the material editor.

Consequences:

Visual Qualities:

- This method provides an easy, fast and cheap way to create pseudo random values.

Performance:

- It is cost efficient as it represents a simple vector based input in the shader graph.

Pipeline (Workflow):

- It is easy to implement.
- Using different object vectors to drive pseudo random values is unstable. For instance, combining the object does destroy this effect.
- Changing the position of an object does change its look, which might be an unintended side effect.

Related Patterns: This pattern can easily be implemented in an *DP 06: Custom Shader* (section 7.3.1.2)

7.8 Summary

The patterns presented in this chapter cover huge areas connected to pattern layering. They provide different possibilities on how to incorporate pattern layering into your pipeline and how to structure and organize the single components. Besides, they illustrate what alternatives exist to pattern layering. This chapter provided different solutions on how to implement the individual components: material container, masking container and blending module and how to utilize different input types. The potential for further work in this field will be presented in the next chapter.

Chapter 8

Discussion

The main goal of this work was to create a catalog of design patterns to support informed decision making in regard to material layering. Therefore, the patterns in this work contain all the necessary information concerning possibilities for application, expected benefits and consequences. They further provide answers to the most important questions of why, when and how to use different pattern layering methods. Most research focuses on new tools and algorithms [4, 19, 23] but fails to provide answers to the questions stated above.

The most important question within this catalog is whether to use a pattern layering method or not. Therefore, two additional non pattern layering design patterns were introduced. The individual patterns were developed and tested during the production of *Letzte Worte* and correspond to my observation in different source files and projects such as [34–40, 43–45].

The research and experiments revealed that pattern layering is often used in production to fit the trends of a modern pipeline. Game production seems to develop towards the following tendencies: the asset production is shifting more into the engine [37, 44], the importance of physically plausible shading and blending is growing, procedural and cooperative pipelines are increasing to enable efficient and iterative workflows. This finding does not correspond to from my earlier expectation of performance as main motivation for using pattern layering.

8.1 Limitations

The design patterns within this work were designed for interactive applications with free player movement, i.e., objects can be seen from different distances and angles. This applies for most first person shooters. The requirements in other applications, for instance with static cameras, may differ from those presented in the catalog. Therefore, the catalog may only be applicable partially. Further, all test were done on PC and with *Unity*[1] and *UE4*.[2] The patterns are supposed to be largely implementation and platform independent. Nevertheless, there might be certain features that are not supported by either the engine or the platform. For instance, an engine might not support all shader

[1]All test cases were realized in *Unity* version 2018.13.f1.

[2]The test cases from *Letzte Worte* use *UE4* version 4.19.2. All other examples use *UE4* version 4.20.3.

inputs discussed in section 7.7 of the pattern catalog and so render parts of the catalog irrelevant. If the engine does not support custom shaders or any built-in material layering solutions, the entire catalog might be irrelevant for that specific case. The catalog does focus exclusively on pattern layering, as other layering methods are not used for video game productions yet. A lot of the patterns might work for other layering methods such as BxDF layering as well, but especially the consequences will be different.

8.2 Conclusion

The design patterns presented in this work provide a practical guide to support informed decision making with regard to pattern layering. Available scientific resources do mainly focus on new technologies, algorithms and tools. Industry specific resources—like documentations, tutorial and articles—do mainly focus on problem specific solutions. They generally fail to explain the long term consequences of your decision making.

This pattern catalog tries to comprehend pattern layering as a universal and largely pipeline independent approach. I have introduced an abstract, implementation independent description model for pattern layering, the *Material Layering Model* which provides the language necessary to do achieve the former goal.

This pattern catalog represents a first step towards simplifying the decision making process of using and creating pattern layering systems and workflow. Future work should focus on automating this decision making process further and transfer the technical complexity from the artist to the software.

Appendix A

List of Design Patterns

References

Literature

[1] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-time Rendering*. 3rd ed. Boca Raton: CRC Press, 2008 (cit. on pp. 5, 6, 8).

[2] Christopher Alexander. *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press, 1977 (cit. on pp. 36, 37).

[3] Christopher Alexander. *The Timeless Way of Building*. Vol. 1. New York: Oxford University Press, 1979 (cit. on p. 36).

[4] Laurent Belcour. "Efficient Rendering of Layered Materials using an Atomic Decomposition with Statistical Operators". In: *ACM Transactions on Graphics*. Vol. 37. 4. ACM. May 2018, 73:1–73:15 (cit. on pp. 12, 16–18, 89).

[5] Brent Burley. "Physically-Based Shading at Disney". In: *Practical Physically Based Shading in Film and Game Production* (Los Angeles). Ed. by Stephen McAuley, Stephen Hill, and Naty Hoffman. Vol. 2012. New York: ACM SIGGRAPH, 2012, pp. 1–7 (cit. on pp. 5, 26, 30).

[6] Christopher Dutton. "Correctly and accurately combining normal maps in 3D engines". *The Computer Games Journal* 2.1 (2013), pp. 41–54 (cit. on p. 26).

[7] Alejandro Conty Estevez and Christopher Kulla. "Production Friendly Microfacet Sheen BRDF". *Technical Report: Sony Imageworks* (2017). URL: https://blog.selfshadow.com/publications/s2017-shading-course/imageworks/s2017_pbs_imageworks_sheen.pdf (cit. on p. 16).

[8] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Boston: Addison-Wesley Professional, 2004 (cit. on p. 17).

[9] Eric Freeman et al. *Head First Design Patterns: A Brain-Friendly Guide*. Newton: O'Reilly Media, Inc., 2004 (cit. on p. 36).

[10] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley, 1994 (cit. on pp. 3, 36, 37).

[11] Jason Gregory. *Game Engine Architecture*. 2nd ed. Boca Raton: CRC Press, 2015 (cit. on pp. 1, 7).

[12] Wenzel Jakob. "layerlab: A computational toolbox for layered materials". In: *SIGGRAPH 2015 Courses*. SIGGRAPH '15. New York, NY, USA: ACM, 2015. URL: https://rgl.epfl.ch/publications/Jakob2015Layerlab (cit. on p. 16).

[13] Wenzel Jakob et al. "A Comprehensive Framework for Rendering Layered Materials". *ACM Transactions on Graphics* 33.4 (2014), 118:1–118:14 (cit. on p. 16).

[14] Jason Jerald. *The VR Book. Human-Centered Design for Virtual Eeality.* Williston: Morgan & Claypool, 2015 (cit. on p. 1).

[15] Brian Karis and Games, Epic. "Real Shading in Unreal Engine 4". *SIGGRAPH Physically Based Shading in Theory and Practice course* (July 2013), pp. 621–635. URL: https://blog.selfshadow.com/publications/s2013-shading-course/ (cit. on pp. 26, 30, 31).

[16] Mike Kime. "Character Art Pipeline for Paragon" (Apr. 2017). URL: https://replay.unrealsummit.co.kr/data/summit2017/unrealsummit004.pdf (cit. on p. 74).

[17] David Neubelt, Matt Pettineo, and Ready At Dawn Studios. "Crafting a Next-Gen Material Pipeline for The Order: 1886". *Physically Based Shading in Theory and Practice* (2013). URL: https://blog.selfshadow.com/publications/s2013-shading-course/rad/s2013_pbs_rad_slides.pdf (cit. on p. 1).

[18] Robert Nystrom. *Game Programming Patterns.* Genever Benning, 2014 (cit. on p. 36).

[19] Colin Penty and Ian Wong. "Gears of War 4: Creating a Layered Material System for 60fps". In: *ACM SIGGRAPH 2017 Talks.* ACM, July 2017, pp. 1–2. URL: https://dl.acm.org/citation.cfm?id=3085026&dl=ACM&coll=DL (visited on 11/06/2018) (cit. on pp. 1, 20, 62, 89).

[20] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering. From Theory to Implementation.* 3rd ed. Burlington: Morgan Kaufmann, 2016 (cit. on pp. 4, 5, 7).

[21] Jenifer Tidwell. *Designing Interfaces: Patterns for Effective Interaction Design.* 2nd ed. Sebastopol: O'Reilly Media, Inc., 2010 (cit. on pp. 3, 36, 37).

[22] Andrea Weidlich and Alexander Wilkie. "Thinking in layers: modeling with layered materials". In: *SIGGRAPH Asia 2011 Courses* (Hong Kong). 20. ACM. New York, 2011. URL: https://dl.acm.org/citation.cfm?id=2077450 (cit. on pp. 4, 14, 27).

[23] Tizian Zeltner and Wenzel Jakob. "The Layer Laboratory: A Calculus for Additive and Subtractive Composition of Anisotropic Surface Reflectance". *ACM Transactions on Graphics (TOG)* 37.4 (2018), 74:1–74:14 (cit. on pp. 16, 89).

Audio-visual media

[24] Allegorithmic. *Material Layering - UE4 shader.* 2016. URL: https://share.allegorithmic.com/libraries/2125 (visited on 11/14/2018) (cit. on p. 59).

[25] Amanda Bott et al. *4.19 Material Layers Preview.* Youtube, Feb. 2018. URL: https://www.youtube.com/watch?v=2dfkedfW1yI&t=1674s (visited on 03/26/2018) (cit. on p. 32).

[26] Clinton Crumpler. *Gears of War 4: DLC Content.* https://www.artstation.com/artwork/4dEOl. Sept. 2016 (cit. on pp. 63, 85).

[27] Epic Games, Inc. *Soul TechDemo*. Youtube, Apr. 2014. URL: https://www.youtube.com/watch?v=jzGRbGb-fog (visited on 10/10/2018) (cit. on p. 51).

[28] Epic Games, Inc. *Soul: City*. Jan. 2018. URL: https://www.unrealengine.com/marketplace/soul-city (visited on 11/14/2018) (cit. on p. 52).

[29] Jacek Maj. *The Witcher 3 Architectural Material*. 2016. URL: https://www.artstation.com/artwork/keXwn (visited on 06/07/2018) (cit. on p. 4).

[30] Harrison Moore. *Paragon Character Texturing Pipeline*. Epic Games, Inc., 2017. URL: https://www.youtube.com/watch?v=nVes6OUyzdw (cit. on pp. 1, 24).

[31] Jeremie Noguer. *The Next Frontier of Texturing Workflows*. Apr. 2016. URL: https://www.allegorithmic.com/blog/next-frontier-texturing-workflows (visited on 06/08/2018) (cit. on p. 2).

[32] Andrew Price. *The Next Leap: How A.I. will change the 3D industry*. Blender Conference. Oct. 2018. URL: https://www.youtube.com/watch?v=FlgLxSLsYWQ&t=1521s (cit. on p. 1).

[33] Jeff Wilson. *Unreal Engine 4.19 Released!* Mar. 14, 2018. URL: https://www.unrealengine.com/en-US/blog/unreal-engine-4-19-released (visited on 10/28/2018) (cit. on p. 63).

Software

[34] CD Project RED. *The Witcher 3: Wild Hunt*. Microsoft Windows PlayStation 4 Xbox One. 2015 (cit. on pp. 1, 3, 4, 56, 89).

[35] Veselin Efremov, Silvia Rasheva, and Torbjorn Laedre. *Book of the Dead*. https://assetstore.unity.com/packages/essentials/tutorial-projects/book-of-the-dead-environment-121175. Jan. 2018 (cit. on p. 89).

[36] Epic Games, Inc. *Infinity Blade: Grass Lands*. https://www.unrealengine.com/marketplace/infinity-blade-plain-lands. Sept. 2015 (cit. on pp. 50, 89).

[37] Epic Games, Inc. *Paragon*. Microsoft Windows, PlayStation 4. 2016 (cit. on pp. 1, 28, 56, 89).

[38] Epic Games, Inc. *Paragon: Agora and Monolith Environment*. https://www.unrealengine.com/marketplace/paragon-agora-and-monolith-environment. Mar. 2018 (cit. on p. 89).

[39] Epic Games, Inc. *Paragon: Crunch*. https://www.unrealengine.com/marketplace/paragon-crunch. Mar. 2018 (cit. on pp. 44, 89).

[40] Epic Games, Inc. *Soul: City*. https://www.unrealengine.com/marketplace/soul-city . Jan. 2018 (cit. on pp. 51–53, 89).

[41] Naughty Dog. *Uncharted 4: A Thief's End*. [CD-ROM]. 2016 (cit. on pp. 1, 56).

[42] Patscheider, Matthias and Povolny, Samantha and Zankl, Bianca. *Letzte Worte*. HTC Vive, Oculus Rift. 2019 (cit. on p. 5).

[43] Ready at Dawn and SCE Santa Monica Studio. *The Order: 1886*. PlayStation 4. 2016 (cit. on pp. 1, 56, 89).

[44] The Coalition. *Gears of War 4*. Microsoft Windows, Xbox One. 2016 (cit. on pp. 1, 56, 62, 89).

[45] Unity, Inc. *Fontainebleau. Photogrammetry Demo Project*. https://drive.google.com/file/d/1qwjyjlG1Ys8engyaPR_XCb3o7Pgz9t5K/view. 2018 (cit. on pp. 54, 55, 86, 89).

Online sources

[46] Allegorithmic. *Substance Painter - Fire Hydrant*. 2017. URL: https://share.allegorithmic.com/libraries/2890 (visited on 11/19/2018) (cit. on pp. 10, 11).

[47] Blender Foundation. *Principled BSDF*. 2018. URL: https://docs.blender.org/manual/en/dev/render/cycles/nodes/types/shaders/principled.html (visited on 05/12/2018) (cit. on p. 30).

[48] Jack Caron. *Normal Map Blending in Unreal Engine 4*. Nov. 2014. URL: http://www.jackcaron.com/techart/2014/11/14/ue4-normal-blending (visited on 11/15/2018) (cit. on p. 26).

[49] Tim Cooper. *Introduction to Shader Graph: Build your shaders with a visual editor*. Feb. 27, 2018. URL: https://blogs.unity3d.com/2018/02/27/introduction-to-shader-graph-build-your-shaders-with-a-visual-editor/ (visited on 01/28/2018) (cit. on p. 32).

[50] Epic Games, Inc. *Asset vs. Instance*. 2018. URL: https://docs.unrealengine.com/en-us/Engine/UI/LevelEditor/Modes/MeshPaintMode/VertexColor/AssetVsInstance (visited on 05/18/2018) (cit. on p. 85).

[51] Epic Games, Inc. *Epic Games Demonstrates Real-Time Ray Tracing in Engine 4 with ILMxLAB and NVIDIA*. Mar. 2018. URL: https://www.unrealengine.com/en-US/blog/epic-games-demonstrates-real-time-ray-tracing-in-unreal-engine-4-with-ilmxlab-and-nvidia?sessionInvalidated=true (visited on 05/10/2018) (cit. on p. 12).

[52] Epic Games, Inc. *Layered Materials*. 2015. URL: https://docs.unrealengine.com/en-us/Engine/Rendering/Materials/LayeredMaterials (visited on 03/10/2018) (cit. on pp. 29, 32, 43, 51).

[53] Epic Games, Inc. *Material Layers*. 2018. URL: https://docs.unrealengine.com/en-US/Engine/Rendering/Materials/MaterialLayers (visited on 11/15/2018) (cit. on pp. 33, 34).

[54] Epic Games, Inc. *Physically Based Materials*. 201. URL: https://docs.unrealengine.com/en-us/Engine/Rendering/Materials/PhysicallyBased (visited on 05/12/2018) (cit. on p. 31).

[55] Epic Games, Inc. *Shader Development*. 2018. URL: http://api.unrealengine.com/INT/Programming/Rendering/ShaderDevelopment/index.html (visited on 10/23/2018) (cit. on p. 31).

[56] Epic Games, Inc. *Shaders and Materials*. 2018. URL: https://docs.unrealengine.com/en-US/Programming/Rendering/ShaderDevelopment (visited on 10/22/2018) (cit. on p. 8).

[57] Epic Games, Inc. *Texture Support and Settings*. 2018. URL: https://docs.unrealengine.com/en-us/Engine/Content/Types/Textures/SupportAndSettings (visited on 11/10/2018) (cit. on p. 25).

[58] Sébastien Lachambre and Sébastien Legarde. *Photogrammetry Workflow Layered shader*. 2017. URL: https://unity3d.com/files/solutions/photogrammetry/Unity-Photogrammetry-Workflow-Layered-Shader_v2.pdf (visited on 10/23/2018) (cit. on pp. 45, 46, 53, 87).

[59] Sebastien Legarde. *Photogrammetry in Unity: Making Real-World Objects into Digital Assets*. Mar. 2018. URL: https://blogs.unity3d.com/2018/03/12/photogrammetry-in-unity-making-real-world-objects-into-digital-assets/ (visited on 11/14/2018) (cit. on p. 55).

[60] Wes McDermott. *The PBR Guide by Allegorithmic - vol. 1*. Apr. 2018. URL: https://www.allegorithmic.com/pbr-guide (visited on 01/21/2018) (cit. on pp. 30, 31).

[61] Wes McDermott. *The PBR Guide by Allegorithmic - vol. 2*. Apr. 2018. URL: https://www.allegorithmic.com/pbr-guide (visited on 01/21/2018) (cit. on p. 5).

[62] Oxford University Press. *surface*. 2018. URL: https://en.oxforddictionaries.com/definition/surface (visited on 06/01/2018) (cit. on p. 6).

[63] Davide Pesare. *MATERIAL LAYERING*. 2017. URL: https://dakrunch.blogspot.co.at/2017/10/material-layering.html (visited on 05/05/2018) (cit. on pp. 12–14, 18, 26, 34).

[64] Mike Seymour. *Epic's State of Unreal + Virtual Human: GDC Day 2: Part 1*. Mar. 2018. URL: https://www.fxguide.com/featured/epics-state-of-unreal-virtual-human-gdc-day-2-part-1/ (visited on 06/10/2018) (cit. on p. 16).

[65] Unity Technologies. *Materials, Shaders and Textures*. 2017. URL: https://docs.unity3d.com/2018.1/Documentation/Manual/Shaders.html (visited on 03/10/2018) (cit. on p. 8).

[66] Unity Technologies. *Shading Language used in Unity*. 2018. URL: https://docs.unity3d.com/Manual/SL-ShadingLanguage.html (visited on 10/24/2018) (cit. on p. 31).

[67] Wikipedia contributors. *Bidirectional scattering distribution function*. 2018. URL: https://en.wikipedia.org/wiki/Bidirectional_scattering_distribution_function (visited on 03/10/2018) (cit. on p. 12).

[68] Wikipedia contributors. *Lookup table — Wikipedia, The Free Encyclopedia*. 2018. URL: https://en.wikipedia.org/wiki/Lookup_table (visited on 11/04/2018) (cit. on p. 9).

[69] Joe Wilson. *Physically-Based Rendering, And You Can Too!* 2015. URL: https://www.marmoset.co/posts/physically-based-rendering-and-you-can-too/ (visited on 03/04/2018) (cit. on pp. 6, 7).

Check Final Print Size

— Check final print size! —

width = 100mm
height = 50mm

— Remove this page after printing! —