

Unreal Engine Substrate: Authoring Materials that matter

Notes have been added to the slides deck because this felt needed since the presentation had to go through a lot of details that may not have been restituted 100% when we did the presentation live. And this is also better for everyone who missed the live presentation.

Introduction / Motivations

- Near State-of-the-art shading models
- Performant - Fortnite runs between 60hz and 120hz
- Consistent across rasterization, ray tracing & path tracing



The current material model in UE is based around the notion of Shading Models. A shading model target a specific set of visual features, proper to a given appearance. For instance, we have a shading mode for general dielectric/conductor, for ClearCoat, for Subsurface scattering, for Cloth, ..

Each shading model is targeting a specific range among the appearance spectrum.

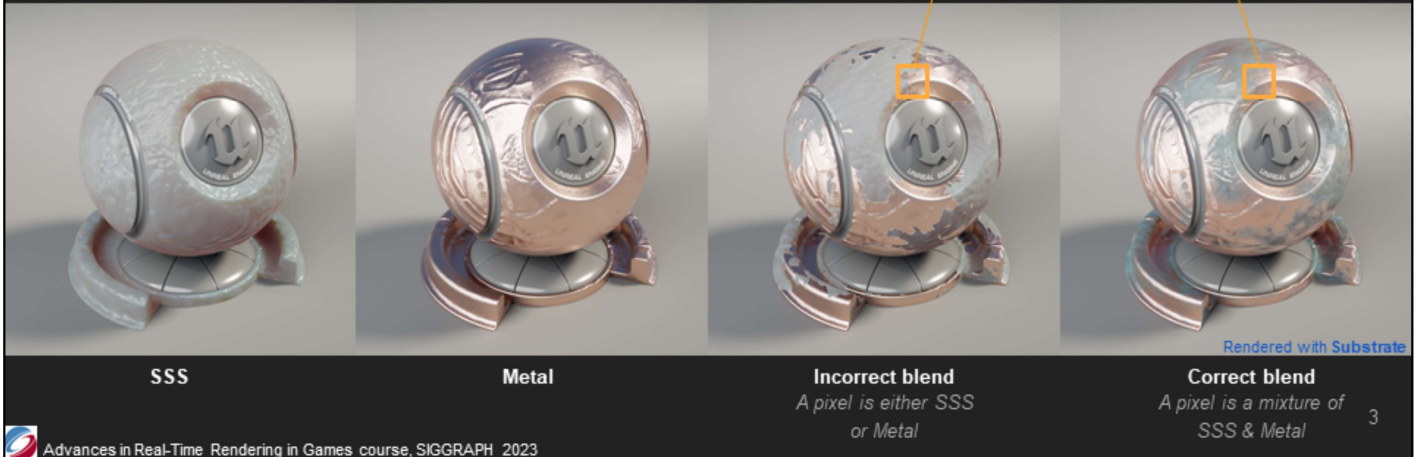
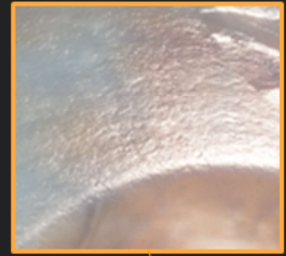
Most of the existing shading model uses state of the art technique / implementation, yet remain performance, since UE is targeting high-performance rendering for game. For instance Fortnite runs up to 120Hz.

In addition all these shading models remain visually consistent across our rendering paths, i.e. rasterization path, ray-tracing effect likeLumen, or our path tracer

Introduction / Motivations

Fixed & Specialized Shading models

*What if artists want to **blend** shading models?*



However, having such specialized shading models, acting in silo, prevents artists from creativity freedom.

For instance what if an artist wants to blend a SSS surface with a metallic surface.

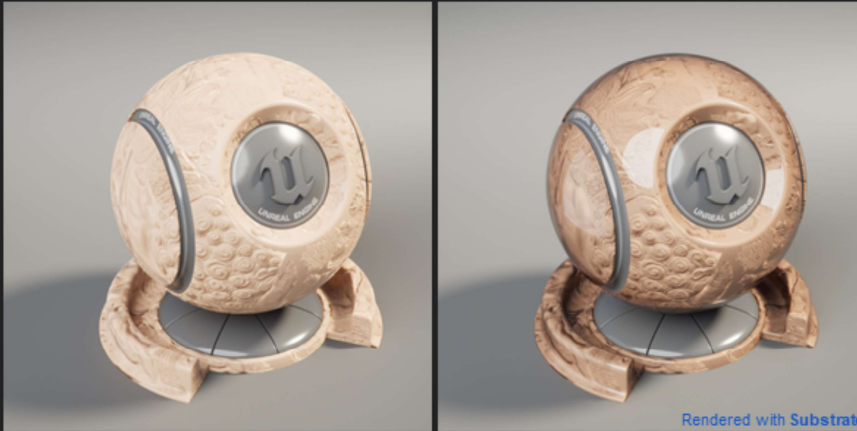
With a shading models approach, since we can represent only one shading model per pixel, the blending would lead to harsh transition/discontinuities.

With proper blending of appearance, the transition becomes smoother, as expected.

Introduction / Motivations

Fixed & Specialized Shading models

*What if artists need a **sub-surface** material with a **coated** layer?*



SSS

SSS with Coat

Another example, what if an artist would like to have a subsurface based material, coated with a plastic armor or some sweat.

Again having a shading model approach, it wouldn't be possible out of the box as it would requires at the same time some feature of our SSS shading model, and some feature of our Coat shading model.

Introduction / Motivations

Fixed & Specialized Shading models

*What if artists need a **two-threaded silk cloth wrapped in plastic?***



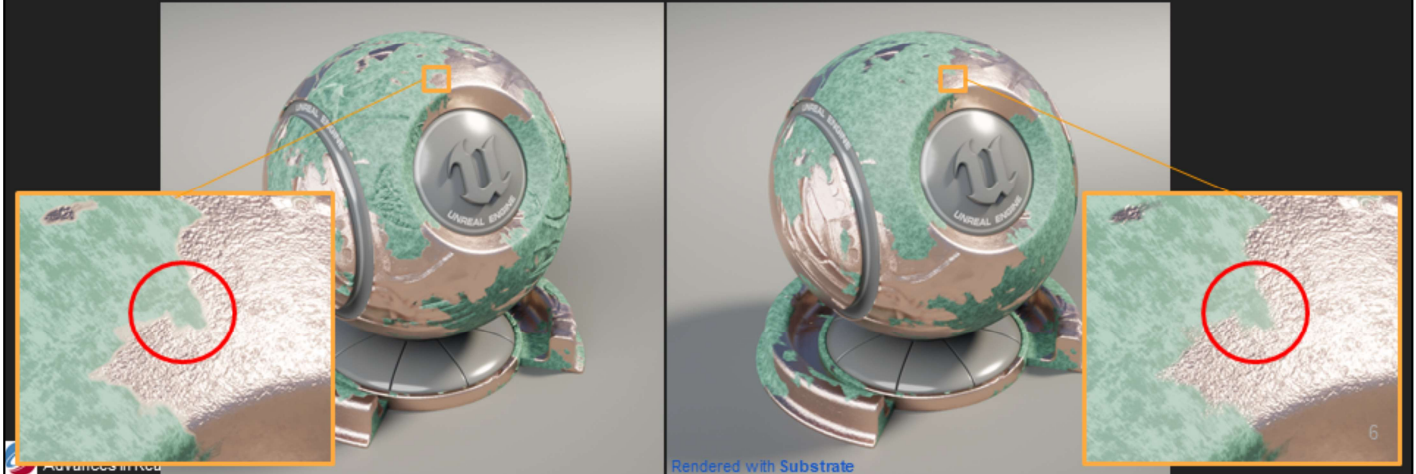
Another more advanced example. Imagine an artist wants to build an advanced cloth material. The material would be made of two sets of silky threads (i.e. with anisotropic highlight), perpendicular to each other to mimic a woven pattern. And the cloth would be covered by a plastic sheet.

Yet again, it wouldn't be possible out of the box with a shading model approach.

Introduction / Motivations

Fixed & Specialized Shading models

*What if artists want to overcome **metalness limitation**?*



Here a limitation of our existing shading models, more tight with our implementation.

UE default shading model uses the notion of metalness to make the distinction between conductor and metallic. When transiting between the two types, an undesired halo can appear. This is a known limitation in PBR, but something we couldn't address before dropping the notion of metalness in Substrate.

Introduction / Motivations

Requires **new** shading models

- **Combinatorial** explosion
- Cost **engineer time** to add new shading models
- Increase **shading cost**
- Features gated by **project settings** (e.g., clear coat dual normal, anisotropy, ...)
- Restricted **input** parameters (*GBuffer memory constraint*)

Cannot import **correctly** some material formats (BSDF soup such as MatX or MDL)

All these examples highlight the fact that, if an artists want to get certain type of advanced appearance, it requires a modification of existing shading models, or even an entirely new shading model.

Adding new shading model each time for such a case would lead to a combinatorial explosion of shading models.

Adding new shading models means engineering cost to implement and ensure everything work as intended.

It also adds, in the long run, an overall increase of shading cost. As more complex shading models are implemented, it is likely that branching & register pressure will increase, dropping performance even for our simple/most used shading models.

To combat these performance regression, one can leverage project settings to only enable these advanced shading models on projects needing them. But that create a maintenance/testing issue.

Furthermore, UE is based on a deferred architecture, which make the number of storable inputs limited.

With all these restrictions, it feels hard to be able to import & support advanced

materials

Introduction / Motivations

Substrate requirements

- Framework for **composing** appearance - create, mix, & layer matter
- As **performant** for existing use cases
- **Scalable** - Visual continuum from mobile to movie quality
- Visually **consistent** between rasterization, ray tracing & path tracing

This is where Substrate comes in light.

Substrate is composition framework, made of building bricks that an artist can assemble to create complex appearance.

Yet, this framework needs to remain as performant as the old system, and also remain visually consistent between our different rendering paths (raster/raytracing effect/pathtracing)

Furthermore, this framework needs to be scalable. Having complex materials is one goal, but we don't want to reauthor variants of the same material for high-end hardware up to for lower-end hardware.

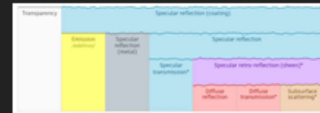
Introduction / Motivations

- Use existing framework?

- Autodesk Standard Surface
- Adobe Surface
- MaterialX
- Renderman Lama
- Nvidia MDL



- Either **Fixed material topology** which is too restrictive



- Or **Soup of BSDFs** which make difficult to have

- Scalability / material simplification
- Efficient evaluation
- Compress/Share parameters

Many material standards & specifications exist in the industry. MaterialX seems to be the one getting the most tracking across the industry.

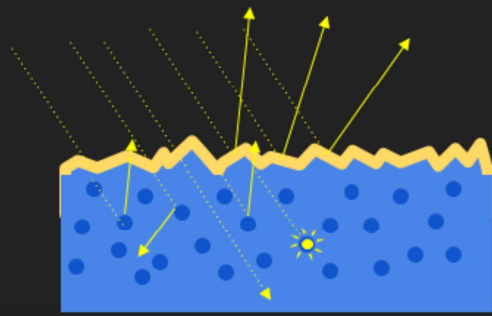
However all these specifications have some limitations which didn't fit our requirements.

Either they rely on fixed topology (Autodesk Std Surface, Adobe Surface, OpenPBR) which prevent to build more advanced surfaces.

Or they rely on a soup of BSDFs, which makes efficient implementation trickier (lot of switches, and loops), and harder to scale material complexity down with simplification rules.

Introduction / Motivations

Substrate



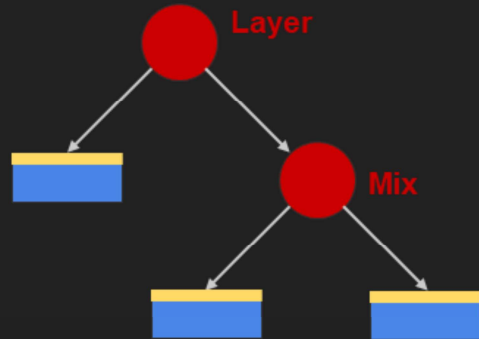
Slab

Participating media + Interface

+ ● =

Operator

Layering, Mixing, Weight



Tree

This is why we started developing Substrate. As said, Substrate is a framework. It is build around three concepts:

- Slab which is how we represent the matter
- Operators which manipulate the pieces of matter
- Tree which represents the topology of the material, i.e., how the different pieces of matter are arranged respectively to each other.

Introduction / Motivations

Dielectric with Absorption



Slab
Operator
Vertical Layering
Slab

Slab Coat
Slab Aniso

Metallic & Anisotropic fiber

To illustrate this, let's take an example.

Here is a material made of two types of matter.

A base layer made of metallic & anisotropic fiber. This is represented by a **Slab**.

Another layer made of a dielectric and having some absorption. This is represented by another **Slab**.

And the two **Slabs** are layered on top of each other with a **vertical layering operator**

Agenda

- Slab
- Operators
- Tree
- Scalability
- Storage & Evaluation

This will be our agenda for today.

Describing the different parts of the framework: Slab / Operators / and Tree.

And then we will dive into details around scalability, storage, and evaluation.

Slab

Defining matter

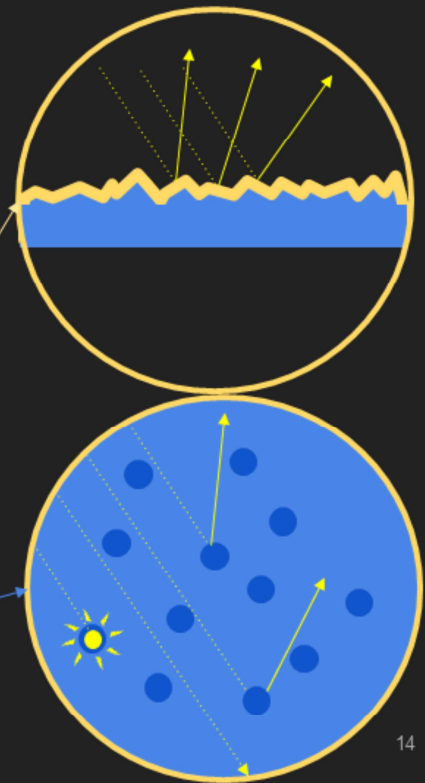
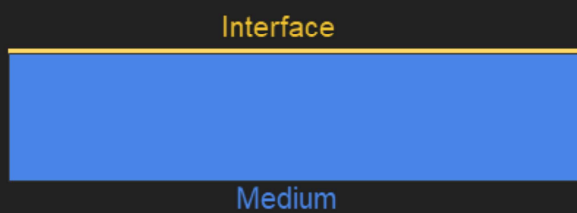
13

Let's start with the Slab

Slabs

Defining matter: Slab

- A **Slab** is a **building block** for describing a piece of matter
As a mean to unify most existing shading models
- A **Slab** is composed of
 - An **Interface**
 - A **Medium**
- Parameterized based on **physical properties**
As physical unit (useful for consistency)



The slab is the building block of the framework for representing a piece of matter. It is our mean to unify most of the existing shading models.

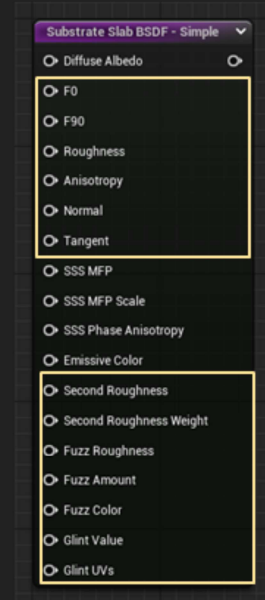
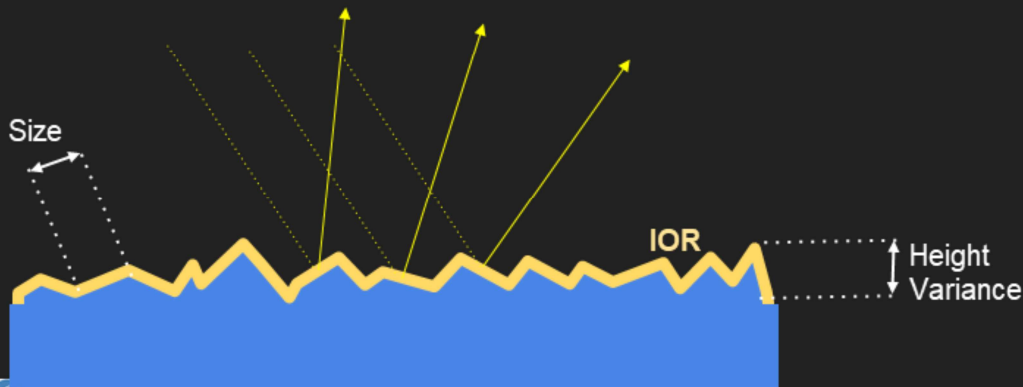
You can really think about as a piece of matter, made of an interface and a medium with a certain thickness.

All the properties are based on physical properties, which helps to describe the matter without too much convoluted reparametrization. This allows to keep things consistent, in particular by using proper units. It helps to resonate/think about what we manipulate and how light would interact when dealing with these properties. A counter example would be gltf, or other specification where diffuse transmission/ color absorption, ... are just different features you manipulation, but does not makes sense as a whole.

Slabs

Defining matter: Slab Interface

- An interface describes **geometry & property** between two media
- Modelled as a field of **microfacet facets**



The interface is as simple as you can think.

It defines the separation between two media, and modelled as a microfacet field. Nothing fancy, the industry's standard are good & simple!

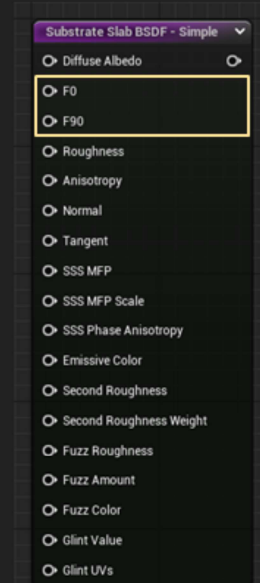
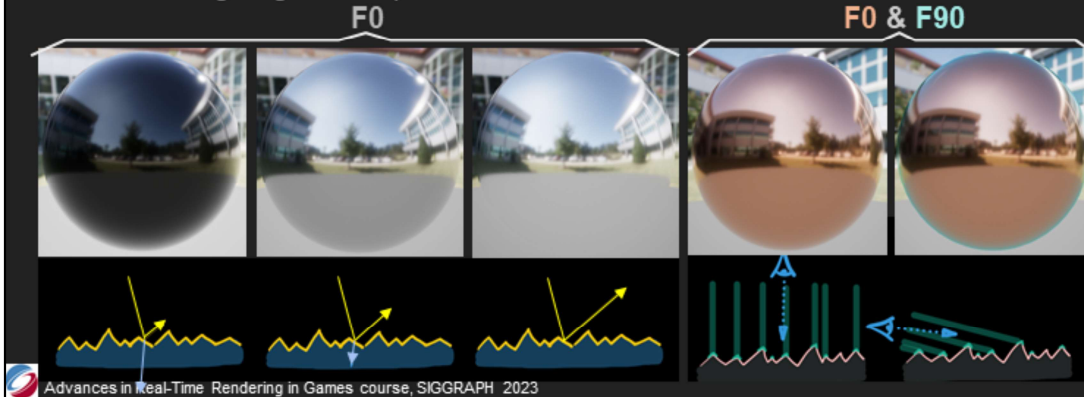
This interface is parameterized with different properties, and we will dive into a few of them.

Slabs

Defining matter: Slab Interface

Reflectivity

- Generic F0 parameterization for convenience (vs. IOR)
- F90 for 'better' match with complex IOR & artistic authoring **Opt-in**
- F90 can only be changed *chromaticity*, not the *intensity*
- Investigating other implementations "Novel aspects of the Adobe Standard Material" Kutz et al. 2021



For describing the reflectivity, we use a F0 parameterization (i.e. reflectivity of the surface a normal incidence) which is convenient since it is normalized unlike (complex) IOR.

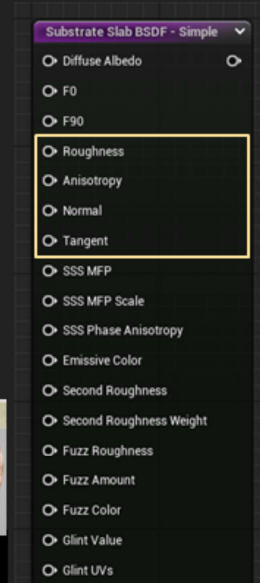
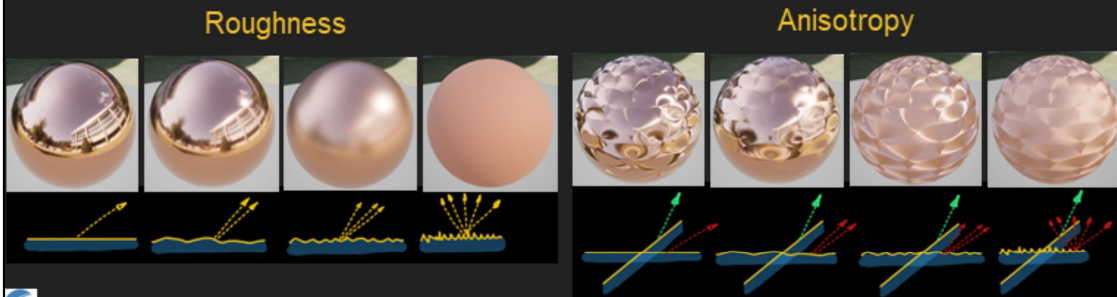
For describing more advanced conductor we used a F90 parameterization, which only affect the chromaticity, not the intensity, do not reduce the energy at grazing handle. This parameterization is the most unphysical one we have, and we might revisit this later in favor Adobe's F82. This features is opt-in meaning you will pay for it only if you used it.

Slabs

Defining matter: Slab Interface

Roughness / Anisotropy

- (Aniso) GGX-based NDF
- Roughness coupled between diffuse & primary specular
- Anisotropy along major/minor axis [-1..1] **Opt-in**
"Revisiting Physically Based Shading at Imageworks" from Chris Kulla & Alejandro Conty, 2017.
- **Adaptive & shared** isotropic/anisotropic tangent basis



For the microfacet description, we use a simple GGX NDF. Again the industry standards are good, so no need to reinvent the wheel.

By default our primary specular roughness is coupled with our diffuse roughness.

For the anisotropy, we use Kulla & Conty parameterization, as it is simple & behave well.

We adapt our tangent basis representation based on usage. If a surface is isotropic we will store a simple normal, but if the surface is anisotropic we will store a full tangent basis with two octahedral encoding (quantized onto 32bits).

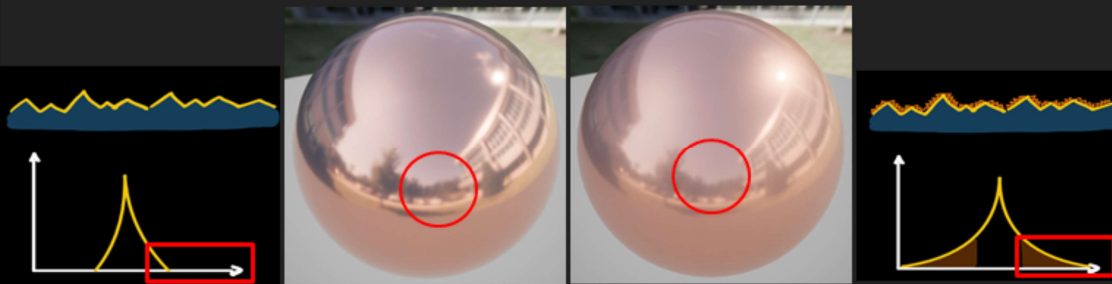
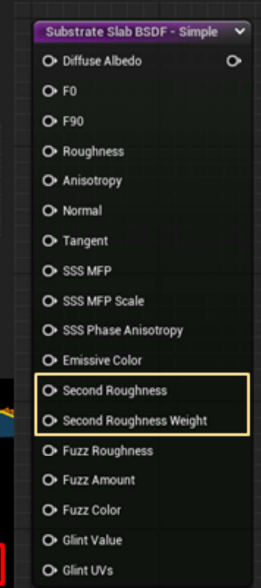
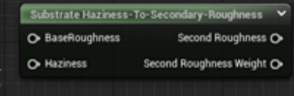
Note: as we will see later, a material can be made of several Slabs, which might share the same tangent a basis and stored only once to save memory. More on this in the Storage section.

Slabs

Defining matter: Slab Interface

Dual specular

- Second GGX lobe mixed with primary **Opt-in**
- Helper nodes for **perceptual mixing**
"A Composite BRDF Model for Hazy Gloss" from Barla et al. 2011
- Side-effect: Specular & diffuse roughness can be **decoupled**



Short tail

Long tail

Maybe a bit more esoteric, we have a secondary GGX lobe, which is mixed with the primary lobe with a free weight.

The primary intent was to represent surface with longer 'specular tail'. A GGX NDF have a certain tail, but for certain manufactured materials, the specular-lobe's tail might be longer. By mixing two specular lobes, we can recreate this hazy appearance, where we can see at the same time sharp reflections and a hazy veil.

We also exposed a helper node for mixing the two lobe in a perceptual manner as showed by Barla et al.

Slabs

Defining matter: Slab Interface

Cloth / Fuzz Opt-in

"Practical Multiple-Scattering Sheen Using Linearly Transformed Cosines"

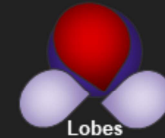
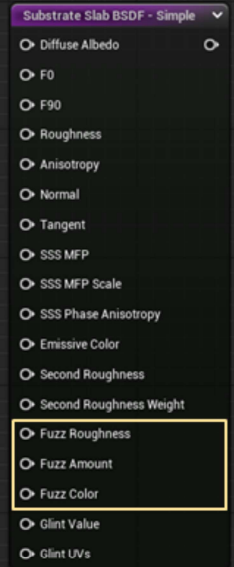
by Tizian Zeltner, Brent Burley, and Matt Jen-Yuan Chiang. SIGGRAPH 2022.

- Always **on top** of specular lobes
- **Separated** roughness & albedo
- Use directional albedo to estimate **transmitted energy**

FuzzAmount = 0



FuzzAmount > 0



A slab can represent asperity on the interface. This asperity layer scatters like in an horizontal manner, unlike specular lobes.

These asperity are always on top of the specular lobes, and have a complete parameterization (roughness, albedo, coverage).

Slabs

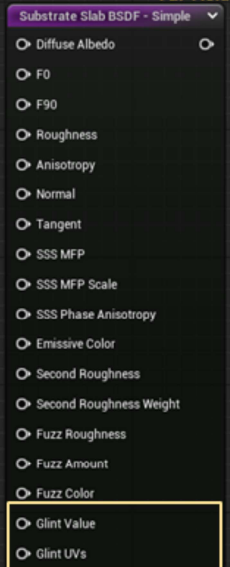
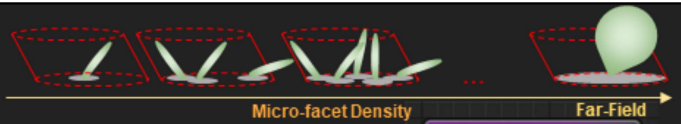
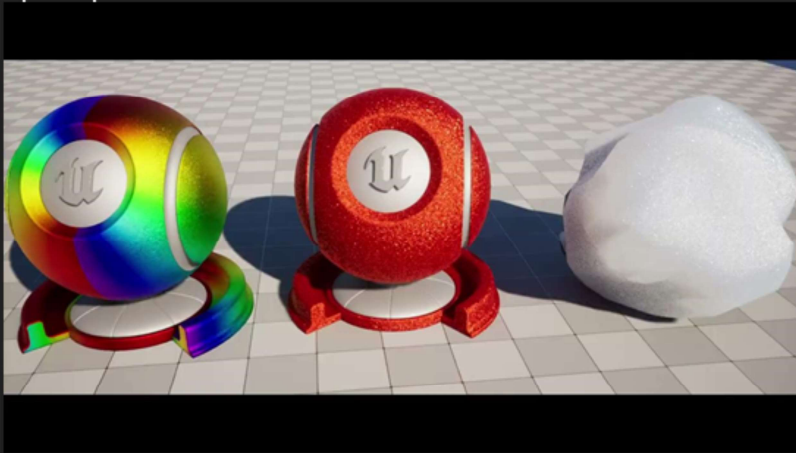
Defining matter: Slab Interface

Glints Opt-in

"Real-Time Geometric Glint Anti-Aliasing with Normal Map Filtering"

by Xavier Chermain, Simon Lucas, Basile Sauvage, Jean-Michel Dischler and Carsten Dachsbacher. i3D 2021.

- Controlled by **glint density** over UV parameterisation
- Scale and filter naturally to **far-field / single-lobe NDF**
- Relies on **precomputed LUT**



As mentioned, we use a microfacet field to model our interface. These microfacets have a certain size and so when we zoom-in on a surface, we should expect to see them. This is where 'glints' comes into play.

You can opt-in to glint and control their size through a density control, parameterized over the surface UV. For achieving this in real-time, we use Chermain et al. approach which uses a precomputed LUTs for defining how the glint behave at different LODs.

For a given 'glint' density, when you zoom-out from the surface, the pixel footprint slowly encompass more & more glints. At a certain distance, it will converge to a single lobe GGX lobe.

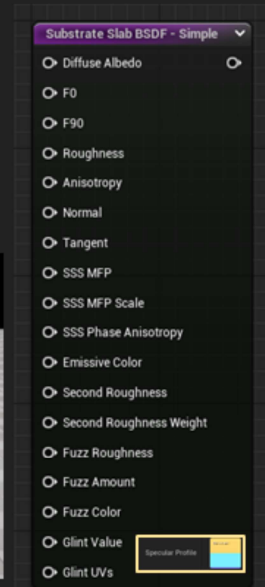
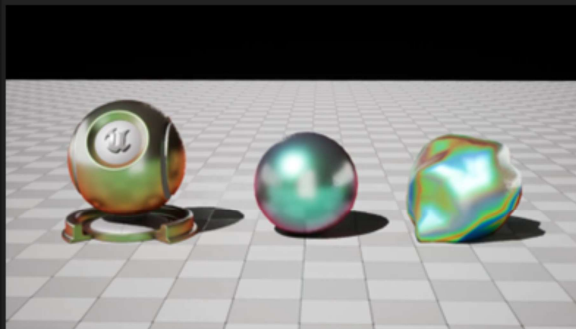
This is mainly used for car paints, snow, or any glittering effects.

Slabs

Defining matter: Slab Interface

Specular LUT Opt-in

- **Advanced** controls on Fresnel - Act as a global 'tint'
- View / Light Parameterization - *Procedural* or *texture* input
- *E.g.*, pearlescence, car paint with view/light correlation



In addition to the reflectivity parameters presented earlier, a Slab can have a 'SpecularLUT' asset plugged-in. This LUT defines a uniform 'tinting' for the slab, parameterized based on view and light angles.

The LUT can be controlled with some ramps or with a texture, parametrized again by view and light angles.

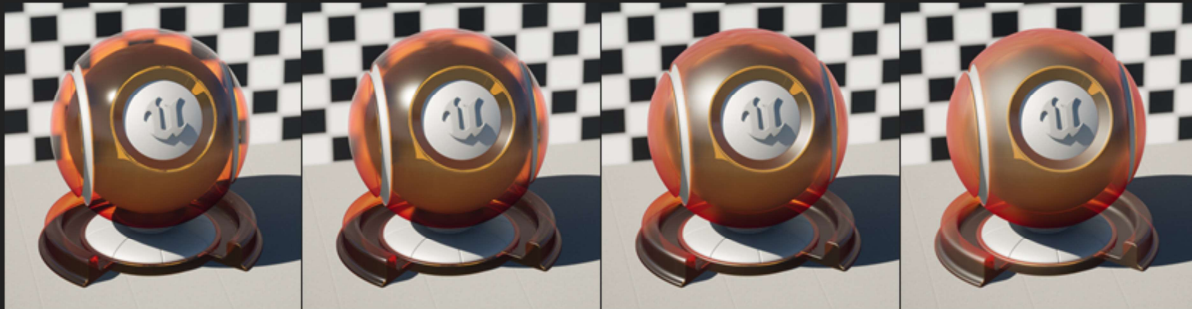
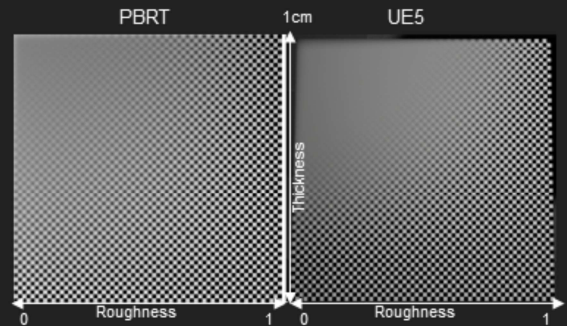
It is useful for advanced cases like opalescence, or complex car paint having view/light correlation, which are not simulated by slab models.

Slabs

Defining matter: Slab Interface

Rough Refraction Opt-in

- Caused by **Rough** interface (*not medium scattering*)
- Refracted ray **defined** by top layer IOR or overridden by user
- Raster uses thickness estimation + LUT



When the interface's roughness is non-null, the light will be scattered. This creates a rough transmission appearance. In this case, the rough appearance is caused by the interface, not the medium scattering (i.e. subsurface transmission)

To simulate this we retrieve the IOR from the interface reflectivity. (A user can be a different methods as well)

For our raster path, we use a precomputed LUT parametrized by roughness and depth/thickness to compute 'light spread'. Then a post-process pass compute the blurring accordingly.

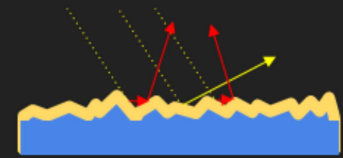
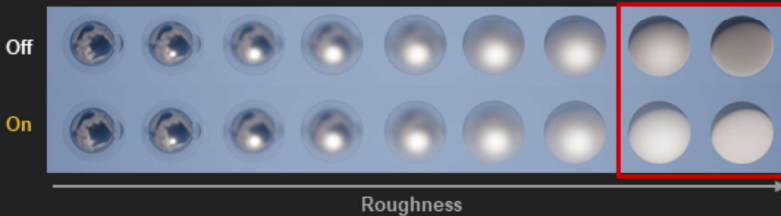
For our RT/PT path, we can simply rely on ray intersection to compute this.

Slabs

Defining matter: Slab Interface

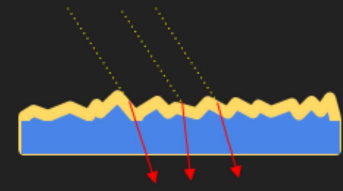
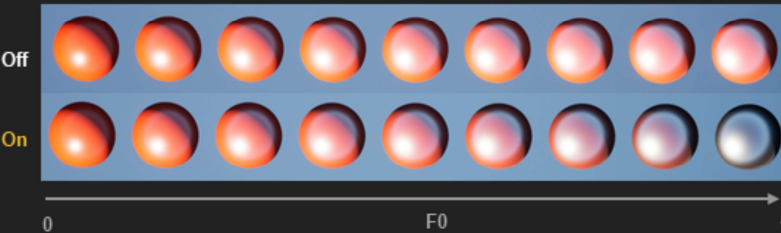
- **Energy conservation:** aka. microfacet multiple scattering

"Practical multiple scattering compensation for microfacet models" from Turquin, 2019



- **Energy preservation:** energy transmitted to the medium

Use precomputed directional albedo to compute transmitted energy



23

One aspect we haven't touched yet is how to manage energy between different parts of the slab.

When an incident light ray hits the interface, it will hit a given microfacet. At that point, it might be reflected, transmitted, or hit another microfacet as it gets reflected

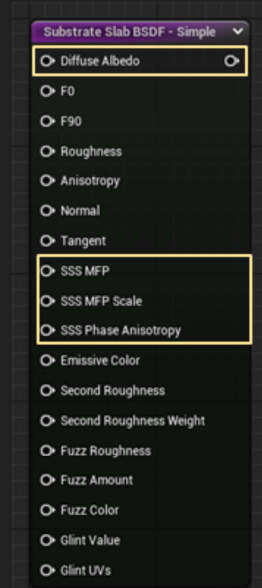
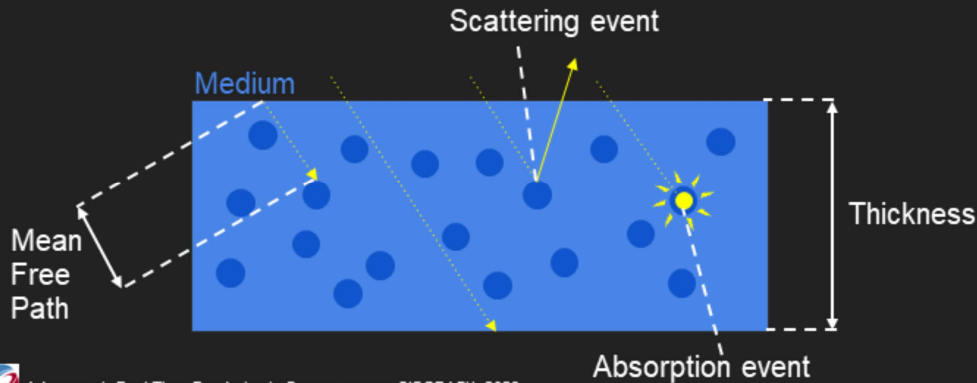
In the latter case, we need to simulate these multiple 'bounces' to not lose any energy. This is very visible at high roughness value, in particular for conductor. To simulate this energy conservation, we rely on Turquin's approach.

The second important aspect is to ensure we are transmitting the correct amount of energy to the medium. I.e. all light which is not reflected & not absorbed by the interface, need to be transmitted to the medium. For this we rely on precomputed table of directional albedo, as described again in Turquin's approach..

Slabs

Defining matter: Slab Medium

- Medium based on **Volumetric formulation**
Allow continuum between traditional Diffuse/SSS/Translucent
- Describe with **Mean-Free-Path / Albedo**
(Dual of Scattering/Absorption coefficients)



This concludes the interface part. Now let's dive into the medium part of a slab.

Our medium is based on a volumetric formulation. The intent was to have a unique parameterization which can represent the continuum between diffuse to fully transparent surface and abstract any artistic parametrization.

As such, a medium is described in terms of:

- Mean free path, i.e. the free-travelling distance of a photon before hitting a medium's particle
- Albedo, i.e. overall medium reflectivity after multiple scattering
- Phase function anisotropy, which describes the scattering directionally when hitting a medium's particle.
- Thickness, i.e. the depth of the medium

Mean Free Path, or MFP for short, and Albedo are just the dual of scattering & absorption coefficients.

Slabs

Defining matter: Slab Medium

“Given a *Mean Free Path* and a *Base Color* (post multiple scattering), Return the scattered luminance for a slab of matter of a given thickness.”

Mean Free Path
Increases \Rightarrow See through

Diffuse Albedo
Increases \Rightarrow Milky

Substrate Slab BSDF - Simple

- Diffuse Albedo
- F0
- F90
- Roughness
- Anisotropy
- Normal
- Tangent
- SSS MFP
- SSS MFP Scale
- SSS Phase Anisotropy
- Emissive Color
- Second Roughness
- Second Roughness Weight
- Fuzz Roughness
- Fuzz Amount
- Fuzz Color
- Glint Value
- Glint UVs

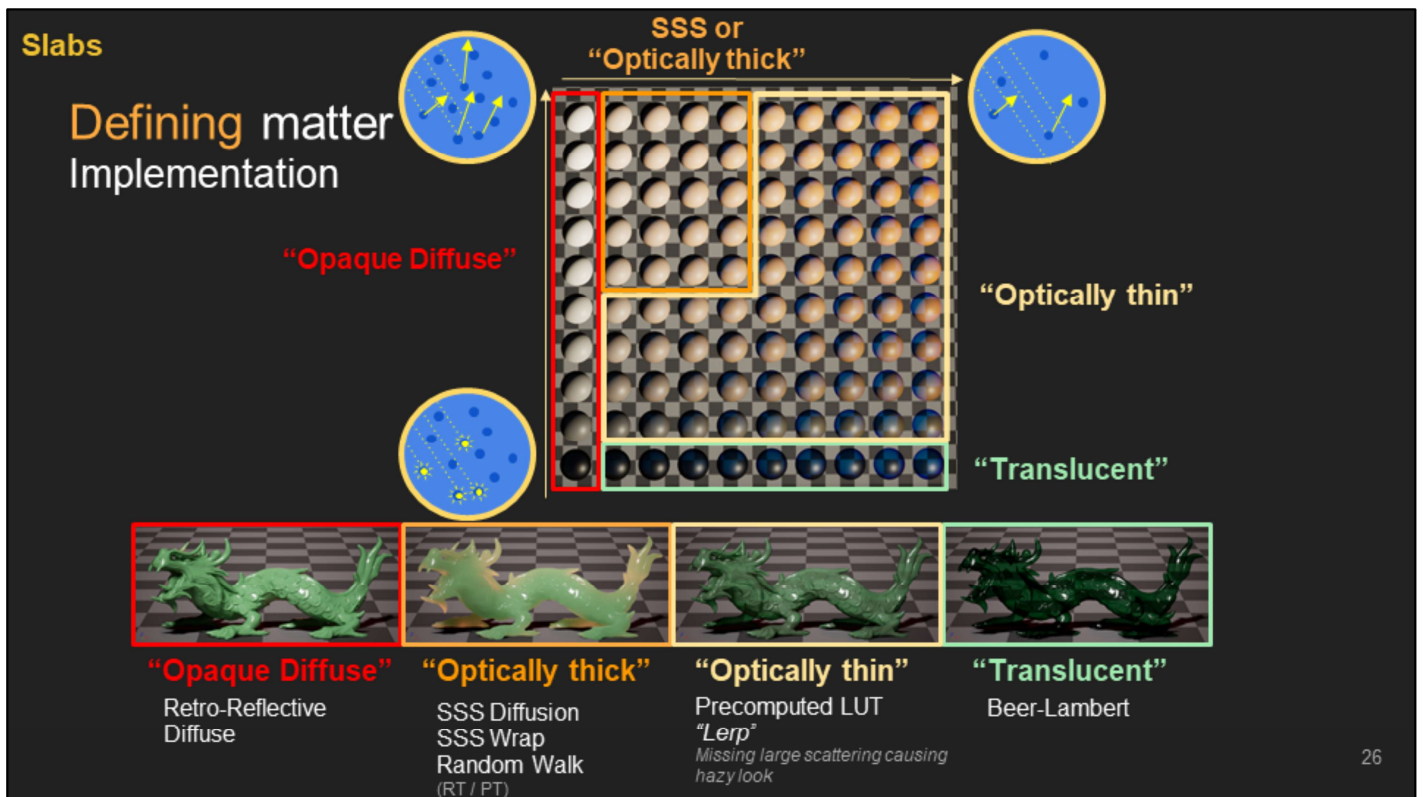
25

Advances in Real-time Rendering in Games course, SIGGRAPH 2023

With such a parameterization we can cover the whole spectrum of appearance. From fully diffuse to full transparent.

In simple terms, you can think of it as:

- Diffuse albedo controls the ‘milky’ appearance
- The mean free path controls how much you can see through the surface




From an implementation point of view, you can divide this space into 4 parts:

- The **opaque diffuse** part, where we can't see through the surface, and no light bleed beyond the pixel footprint
- The **optical thick** part where, you can't see through the surface, but some light bleed beyond the pixel footprint
- The **optical thin** part where you can see through the surface, but there are still some scattering happening
- The **translucent** part, which is the special case of optical thin part, without any scattering.

Let's dive into how we support/implement each of these parts.

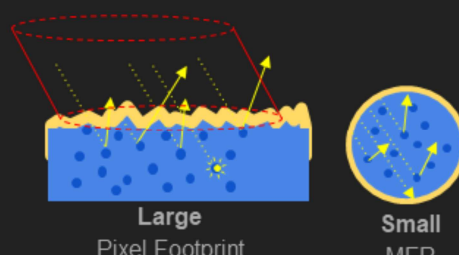
Slabs

Defining matter: Slab Medium

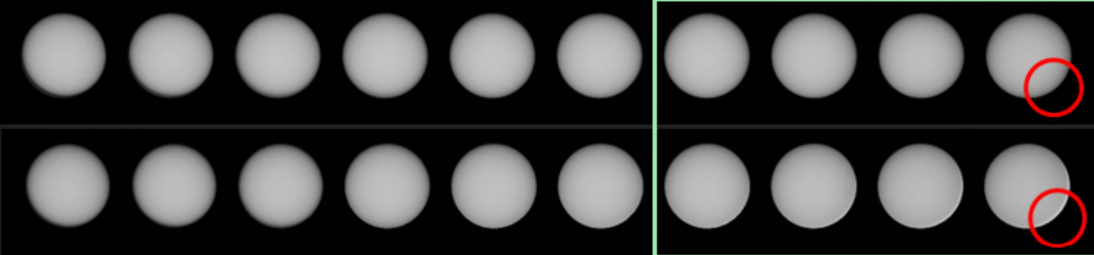


Retro-Reflective Diffuse **"Opaque Diffuse"**
"Material Advances in Call of Duty: WWII" from Chan 2018

- Parameterized by roughness
- Roughness coupled with primary specular lobe
- Use for small MFP or large pixel Footprint



Lambert Diffuse



Chan Diffuse
 Retro-reflective at high roughness

Advances in Real-Time Rendering in Games course, SIGGRAPH 2023

27

For the diffuse part, we use a retroreflective model, based on Chan.

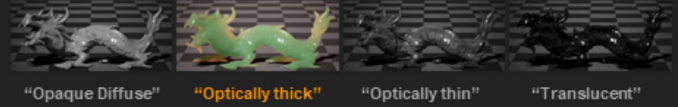
This approximates a diffuse microfacet model, where the roughness is coupled with the primary specular.

The roughness controls the amount of retro reflection, i.e. the light bouncing back at grazing angle. This is particularly visible at high roughness.

As per our diagram earlier, this 'regime' is used when the pixel footprint is large compared to the MFP.

Slabs

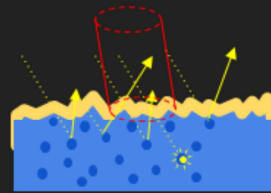
Defining matter: Slab Medium



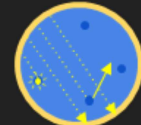
SSS Diffuse / Wrap "Optically thick"

"Real-time subsurface scattering with single pass variance-guided adaptive importance sampling" from Xie et al 2020

- Raster uses *screen-space diffusion* algorithm
- RT/PT uses *random walk* algorithm
- Applied only on the **bottom** layer
- Use wrap lighting cheap/legacy purpose (*opt-in*) or lower platform
- Use for small MFP or large pixel Footprint

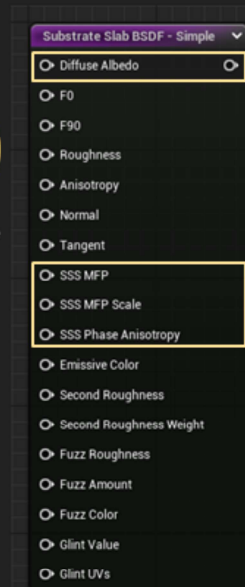


Large
Pixel Footprint



Med./Large
MFP

Mean Free Path



28

When this 'regime' is used when the pixel footprint is smaller compared to the MFP, we enter into the Optically thick regime

For this we use regular SSS techniques.

- For our raster path we use a post process method based Xie et al.
- For our RT/PT paths, we use random walk algo.

Note that, in our implementation, if a material is made of several slabs, this is applied only on the most bottom visible slabs.

For the raster path, we can fallback on other method like wrap lighting for performance reason.

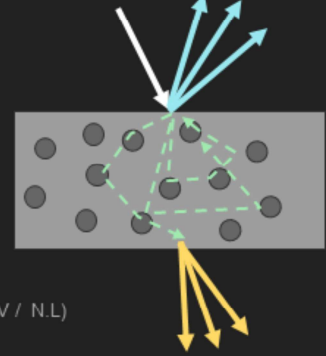
Slabs

Defining matter: Slab Medium



Custom LUT-Based scattering "Optically thin"

- Scattering back and transmitting energy
- Used mainly for **optically thin** layer on top of **opaque** layer
- Volumetric effect *approximated* as a **BSDF**
Future Work → Handle spatial large scattering causing hazy look



Our model

- Scattered back energy → **Precomputed 4D LUT** (Single Scattering Albedo / Rescaled MFP / N.V / N.L)
- Transmitted energy → Beer-Lambert absorption

Thickness of scattering layer →



29

The optically thin case is rather uncommon in real-time rendering as it is rather hard to handle. It combines at the same time some translucency (you can see through) and some large-scale scattering (hazy look).

In our case, we focused on the special case of **an optically-thin slab stacked/layered on top of another slab**.

In such a regime, there are two parts: the **light scattering back**, and the **light transmitted** through the surface.

- For the transmission part we use a simple Beer Lambert model for the absorption.
- For the scattering, we rely on precomputed LUT, parameterized by single scattering albedo (estimated from the Diffuse Albedo), rescaled MFP (rescaled for 1m thick slab), and view & lights angles. This LUT was computed offline by computing the luminance scattered back by a piece of medium under various MFP/albedo/view/light conditions.

This approaches only solves the first moment, i.e. the amount of energy, but does not handle the 'spreading' which cause the hazy look of such a surface. We leave that for future work.

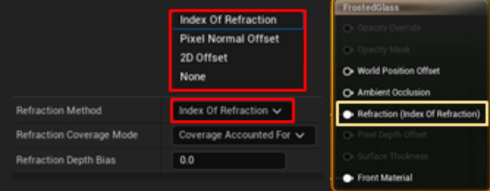
Slabs

Defining matter: Slab Medium



Beer Lambert “Translucent”

- Simple absorption for non-scattering material ($DiffuseAlbedo=0$)
- Beer lambert model - Use *dual source blending* for correct composition
- Refraction uses material IOR or custom user input (to avoid raster artefacts)
- Decoupling of **coverage & transmittance** (see later)



Finally the special case of optically-thin surface, where we can ‘see through’ but doesn’t have any scattering.

We simply handle this with a beer-lambert approach. For our raster path, we use dual source blending to properly handle the transmission part. Again nothing fancy, just industry standard.

The important part here is that we decoupled the notion of transmission from the notion of coverage. This split the convoluted ‘opacity’ term into two explicit notions. More on that later.

Slabs

Defining matter: Slab Medium

MFP vs. Pixel Footprint / Thickness

- Equivalent in appearance
- Importance of **units**! MFP is expressed in **meter**
- Allow to switch SSS on/off in distance (*only used for SSS pass*)



Small MFP



Large MFP

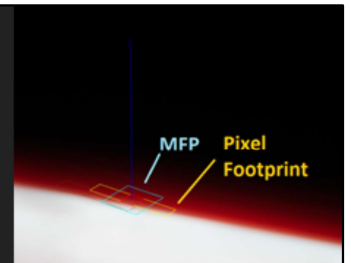
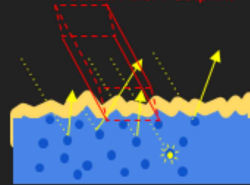
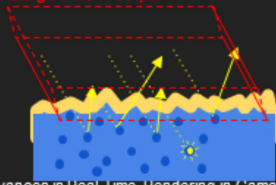
Mean Free Path



Large Pixel Footprint

Pixel Footprint

Small Pixel Footprint



As we have seen, the **scale** at which a surface is looked at, dictates what regime should be used. A surface with a given MFP value looked from far away will be rendered as diffuse, while same surface looked at closely, will be rendered as optically-thick with some subsurface scattering.

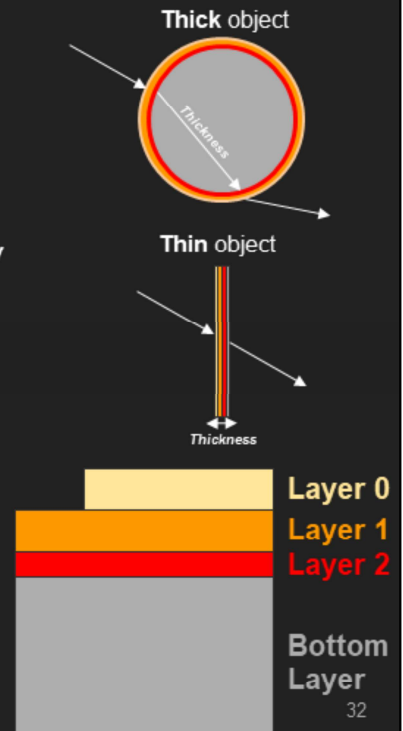
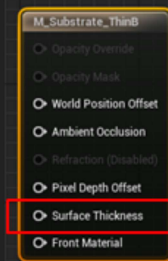
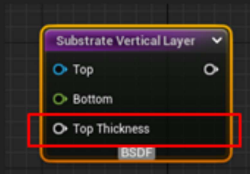
I really want to emphasize that **units** are important. Having pixel footprint and MFPs expressed proper world units help to know how to render a surface.

Slabs

Defining matter: Slab Medium

Thickness

- **Bottom layer**
 - **Thick surface:** geometry is non flat. The thickness is defined by the **geometry**
 - For **Raster** → thickness = hardcoded to 0.1cm
 - For **Ray/Path Tracing** → thickness = ray distance
 - **Thin surface:** geometry is flat. Thickness is defined by the **graph**
 - Set on the root node for the entire graph
- **Other layer**
 - Defined by **Vertical** operators (*more details later*)
 - Uses the **Translucent** medium model



Finally, the thickness of the medium need to be taken into account. It defines the distance travel by the light path when crossing a medium, which impact 'the light attenuation'.

When there is a single layer or the most bottom one (if there are multiple layers): we need to consider two cases

- If the surface is considered 'thick', the thickness is defined by the geometry's boundary. For the raster path, we can use a constant value, a SDF or a shadow map to estimate the thickness. For the RT/PT, this is given by the distance to the ray's intersection.
- If the surface is thin, i.e. the geometry is a flat surface (e.g. foliage, cloth, ...), the thickness needs to be defined in some way. In such a case we define the thickness at the material level, with a Surface Thickness input on the root node

For other layer, we also define the thickness through the graph. This is only necessary when a slab is 'coating'/'layered on top of' another slab. In such case, the thickness of the top slab is provided by the Vertical operator.

Operators

Manipulating matter

33

This concludes the Slab section. Let's now dive quickly onto the operators to manipulate the matter.

Operators

Manipulating matter - 3 operators

Slab Slab

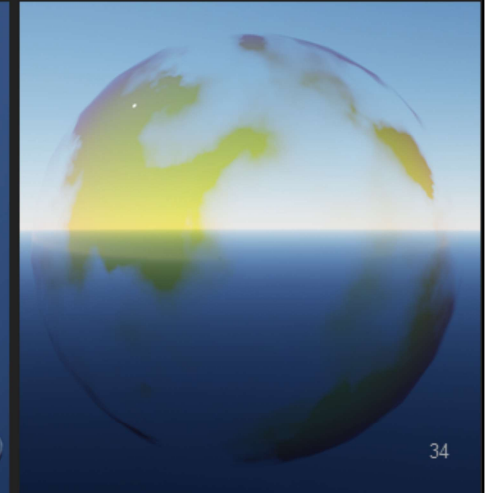
Horizontal mixing - aka lerp

Slab
Slab

Vertical layering - aka coating

Slab

Coverage weighting - aka alpha



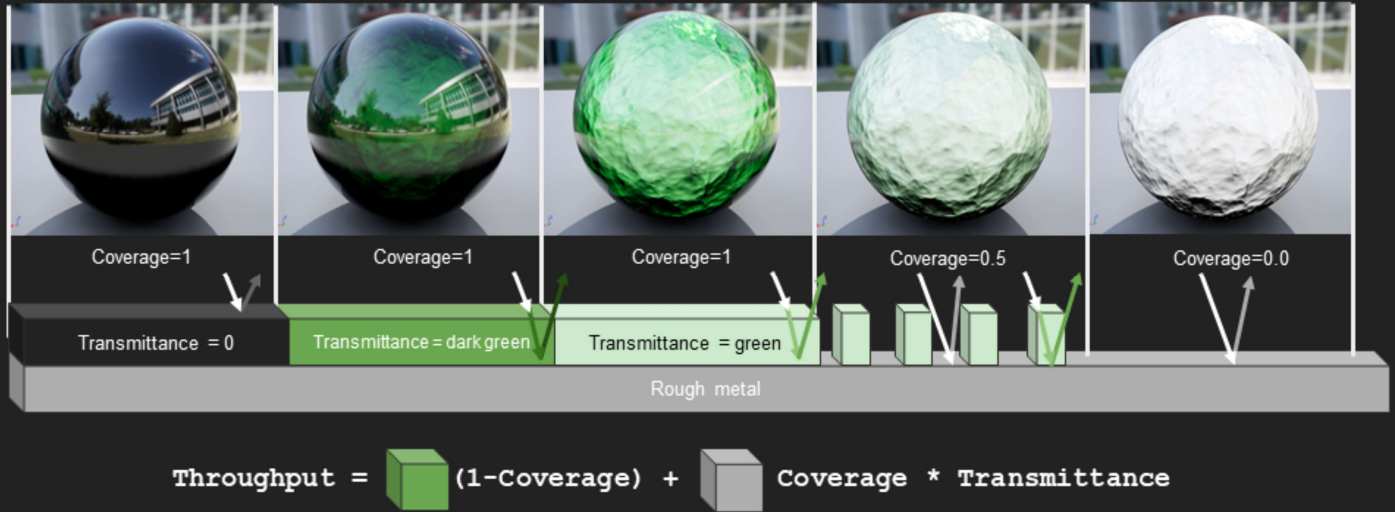
34

We have mainly 3 operators.

- Horizontal mixing - think of it as a lerp/blending between two materials
- Vertical layering - think of it as a coating/layering between two material
- Coverage weighting, that you can think of it as alpha blending, to make a material more 'transparent'

Operators

Terminology - Coverage, transmittance & throughput



But first, let's define properly our terminology to ensure we are talking about that same thing

Let say we have a rough metallic sphere at the base. When the light hits the surface, it reflects directly from this rough metal.

Now, let say we coat it with a greenish dielectric material.

On the left most part, the thickness of the material is so thick, that when the light hit the surface, it only reflects from the dielectric layer. All the transmitted light is absorbed by the dielectric layer. In such a case, **the transmittance of the dielectric layer is 0**

If we reduce the thickness of that dielectric material the transmitted light will be absorbed but only partially. And so the the light will be able to reach the rough metal beneath. In such a case, the **transmittance of the dielectric layer start to be >0**

If we reduce the thickness even more, then less light will be absorbed, and so **the transmittance will increase even more.**

In the first three cases, the dielectric layer is always there, i.e. if the light travel from the camera to the surface, it will hit 100% the dielectric layer. **The coverage of the dielectric layer is 1**

On the other hand, on the last case, the dielectric layer was never there. **The**

coverage of the dielectric layer is 0

By extrapolating, having a **coverage of 0.5** means that you have 50% of change of hitting 'the dielectric layer and then rough metal surface', and 50% of change for hitting only the rough metal surface.

So coverage defines the probability of presence of a slab, while the transmittance defines 1-attenuation of a slab

Combined, the **coverage** and the **transmittance** of a slab defined its **throughput**

Now let's dive into the operators

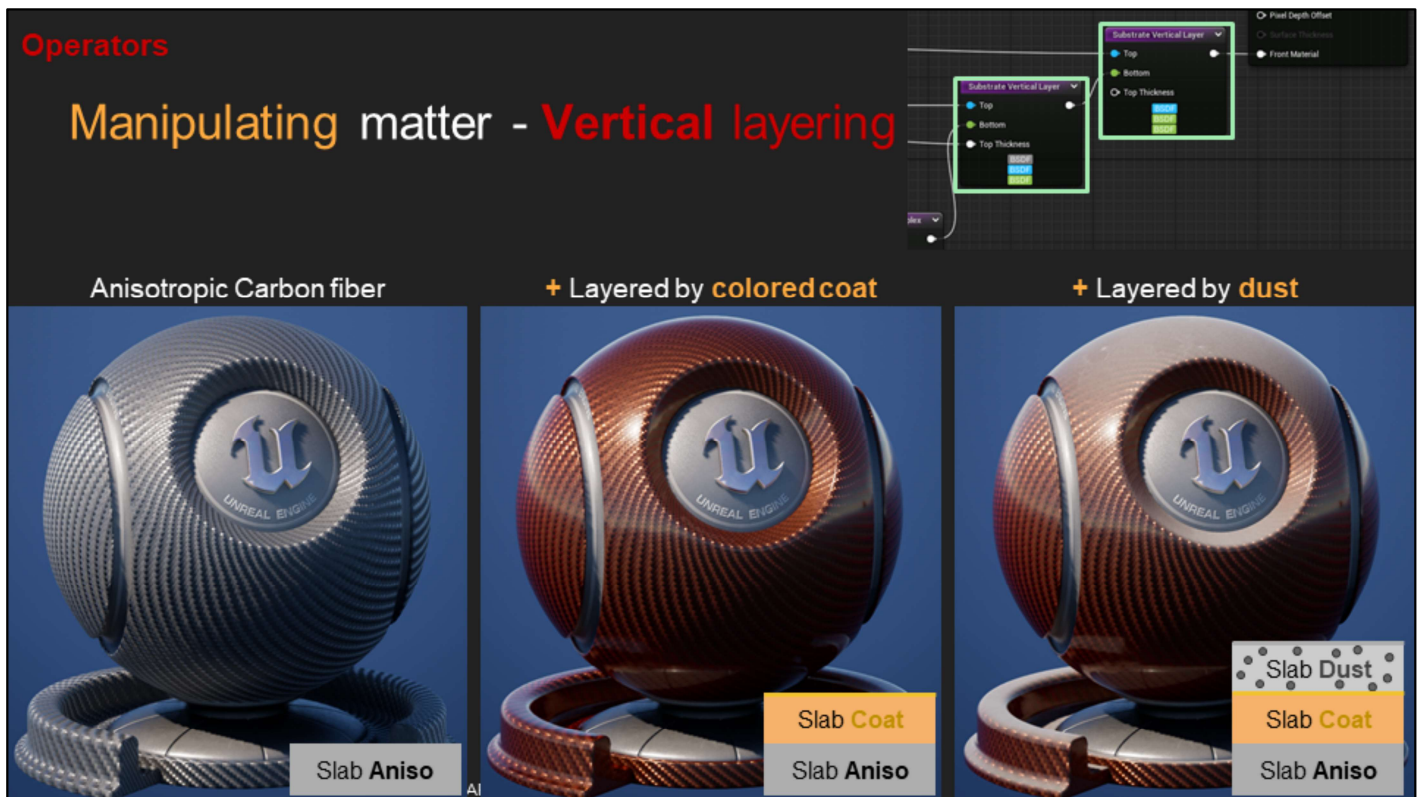
Operators

Manipulating matter - Horizontal mixing



The horizontal mix operator blends materials.

Let's say we have a gold material, and we 'horizontally mix' it with a SSS material: it means that we will evaluate both materials, and mix their evaluation. The mix factor defines the coverage of the slabs. The first slab will have a coverage of 'mix', while the second slab will have a coverage of '1-mix'



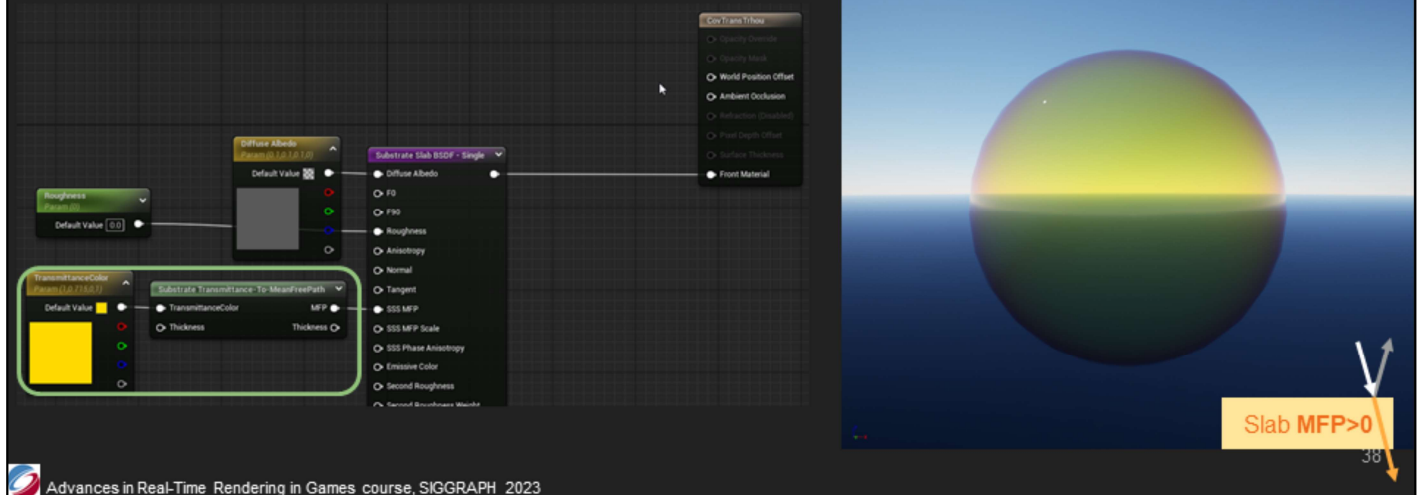
The vertical layering operator, allows to stack a slab on top of another one.

Let's say we have a aniso fiber material. We want to coat it with a dielectric material. By doing so the **transmittance** of the dielectric material will affect the appearance of the anisotropic material, given it a reddish tint. Layered operations can be combined. For instance we stack another dust layer on top of it, which again will affect transmittance and throughput of the bottom layers.

Operators

Manipulating matter - Coverage weight

Translucent sphere / $MFP = \text{Yellow transmittance}$



Lastly, the coverage weight operator.

We start first with a dielectric with some absorption. The light traveling through the surface get partially absorbed, giving this yellowish **transmittance**.

Operators

Manipulating matter - Coverage weight

Translucent sphere / MFP = Yellow transmittance / Varying coverage



Advances in Real-Time Rendering in Games course, SIGGRAPH 2023

By weighting such a surface with a coverage operator we can affect the coverage of the slab, i.e. the amount of 'presence' for a given pixel.

- If the coverage is 1, we completely see the surface
- If the coverage is 0, we don't see the surface
- If the coverage is in-between we see a portion of that surface.

Operators

Manipulating matter - Coverage weight

Rough metal + Smooth colored coat



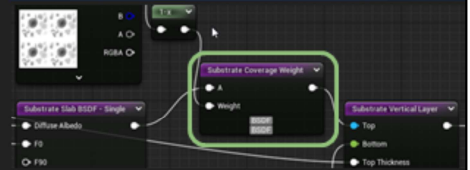
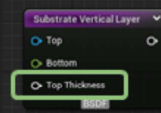
Coat + normal + **varying thickness**



Coat + normal + **varying coverage**



Thickness==0
maintains Coat specular



It is important to have this clear distinction between **varying thickness** and **varying coverage**.

Let's say again, we have a metal surface coated with an orange dielectric.

With a **vertical layering operator**, when we vary the thickness of the top layer, we affect both the transmittance & the throughput of the top slab, but we can always see its specular reflection

While a **coverage weight operator**, when we vary the coverage of the top layer, the top slab transmittance remains the same, only its throughput vary (accordingly to the coverage). But when the coverage reaches 0, we can no longer see the specular reflection.

Substrate Tree

Organizing matter

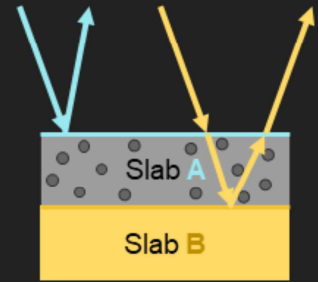
41

So as you can see, one can create material using slab of matter assembled using operators. But how do we organise such data in order to be able to evaluate the lighting at the end?

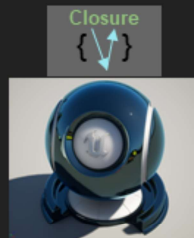
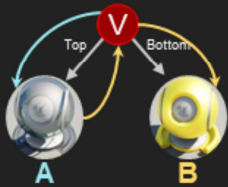
Tree

Substrate Tree

Represent material topology



Substrate Tree ⇒ list of closures



Closures = parameters sent to the renderer for lighting evaluation, accounting for topology, view & light path.

42

First let's say you have want to render such a yellow piece of rough metal covered with a clear coat feature blue specular.

In term of Substrate, this would be represente by a slab B for the rough metal, with a slab A covering it, achieved using a vertical layering operator.

But we cannot evaluate those two slabs directly and add the two lighting contributions. That would be wrong because we need first to evaluate the effect that slab A has on slab B when it comes to throughput accounting for the medium transmittance or fresnel effect for instance.

In order to account for such effect, **we represent the material topology using a tree**, operators being nodes and slabs being leaves.

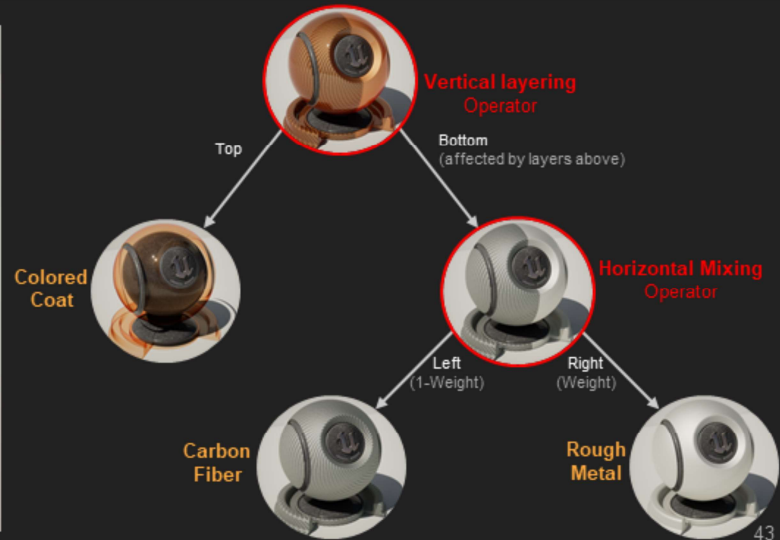
We are going to process/walk/evaluate this tree in order to be able to output closures. Closures represent a bag of parameters that can be sent to the renderer for lighting evaluation, accounting for all the different operators effects on the visual result. In this case, we can see that Slab A is at the top of the material topology, so the lighting can be evaluated as is. While we see that slab B is at the bottom of the hierarchy, so we need to account for the light path through SlabA, and for instance its effect on throughput and perceived roughness, in order to be able to evaluate Slab B.

Once evaluated, a Slab outputs a closure. Each of those closures can be evaluated in any order or in parallel. Each Closure evaluation output a luminance color, all of them can be summed up together in order to form the final image.

Tree

Substrate tree represents material topology

Another example:



Let's have a look at a more complex example.

Here we have carbon fiber, horizontally mixed with a rough metal. You can see a transition region between both slabs. That is why those two slabs are linked with each other using an horizontal mixing operator.

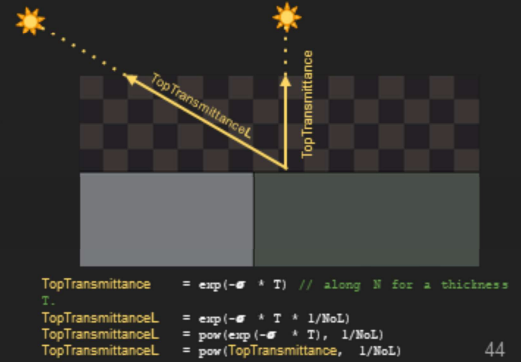
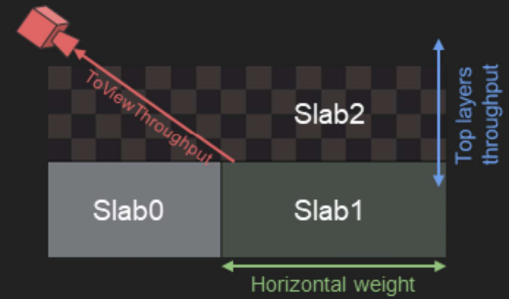
And on top of that, you can see a colored clear coat that is layered on top of both the other slab using a vertical layering operator.

Tree

Walking the Substrate Tree

Used to evaluate many quantities:

- **ToViewThroughput**
 - Horizontal mixing weight
 - Top layers throughput (Using coverage, specular transmittance, medium transmittance)
 - Coverage weight
- **TopTransmittance**
 - Transmittance at normal incidence due to layers above
 - Lighting passes: evaluate for each light w.r.t. **N.L**
- **Rough refraction data**
 - Using roughness and thickness



From this tree we are going to evaluate many closure quantities so that we can evaluate the effect that each operators and slabs have on each other.

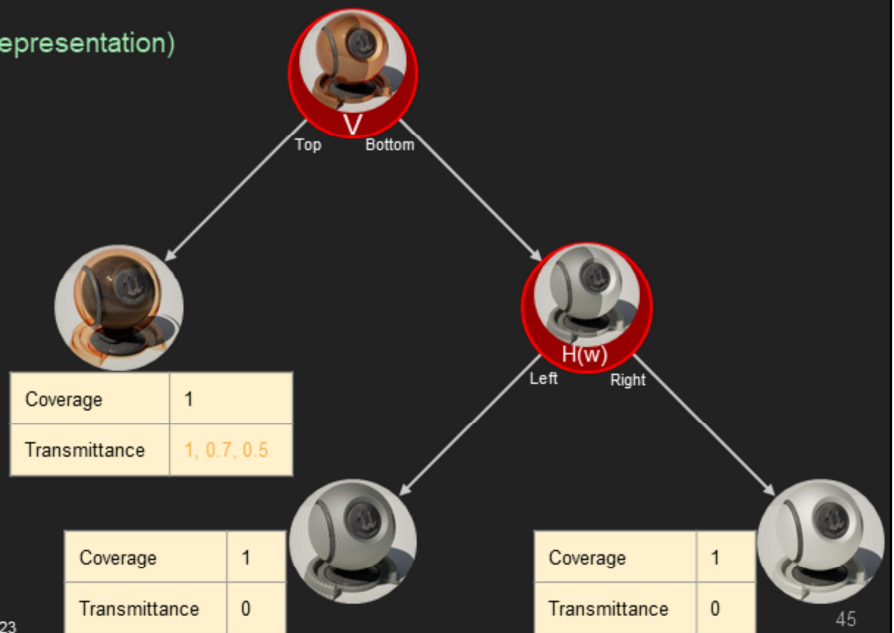
For instance

- The **ToViewThroughput**, being affected by horizontal mixing weight, top layers throughput, or coverage weight.
- The **TopTransmittance** representing the transmittance of all the surfaces above a given slab, that we can reuse when evaluating the lighting, remapping it to the actual light direction using a simple formula.
- We also output **roughness and thickness** data used for rough refraction effect (detailed later on)

Tree

Walking the Substrate Tree

1- Initialise Slabs C/T (statistical representation)



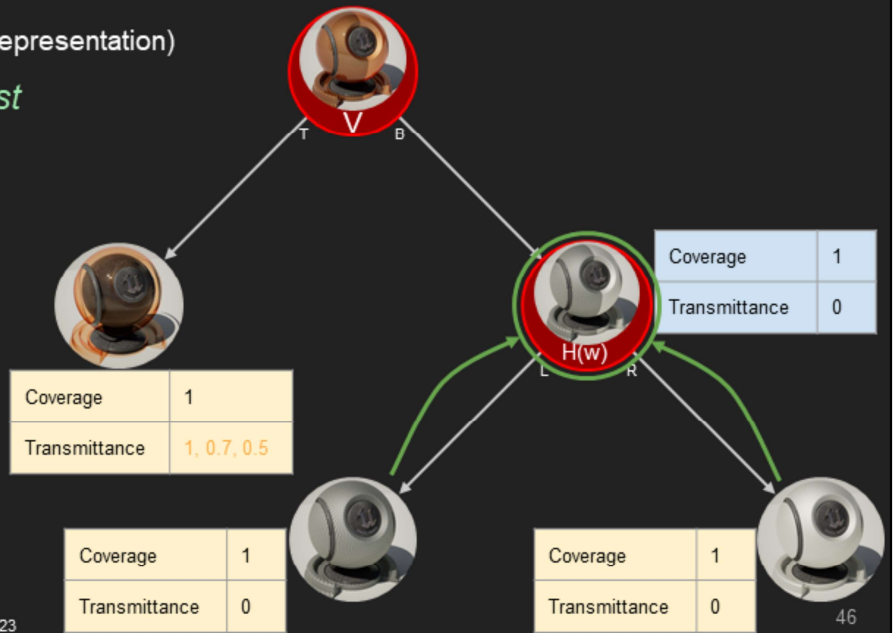
So how do we process and walk the tree? To remain simple on those slides, we are only going to look at how we update the Coverage and Transmittance of a slab that will be used by the closures for lighting evaluation.

Firstly, we evaluate the coverage and transmittance from each slab. You can see that both slabs at the bottom are opaque so coverage is 1 and transmittance is 0. The colored coat slab has coverage of 1 and a transmittance mapping resulting from the mean free path as seen when viewing the material in isolation along the normal of the surface.

Tree

Walking the Substrate Tree

- 1- Initialise Slabs C/T (statistical representation)
- 2- Process operators, *depth-first*



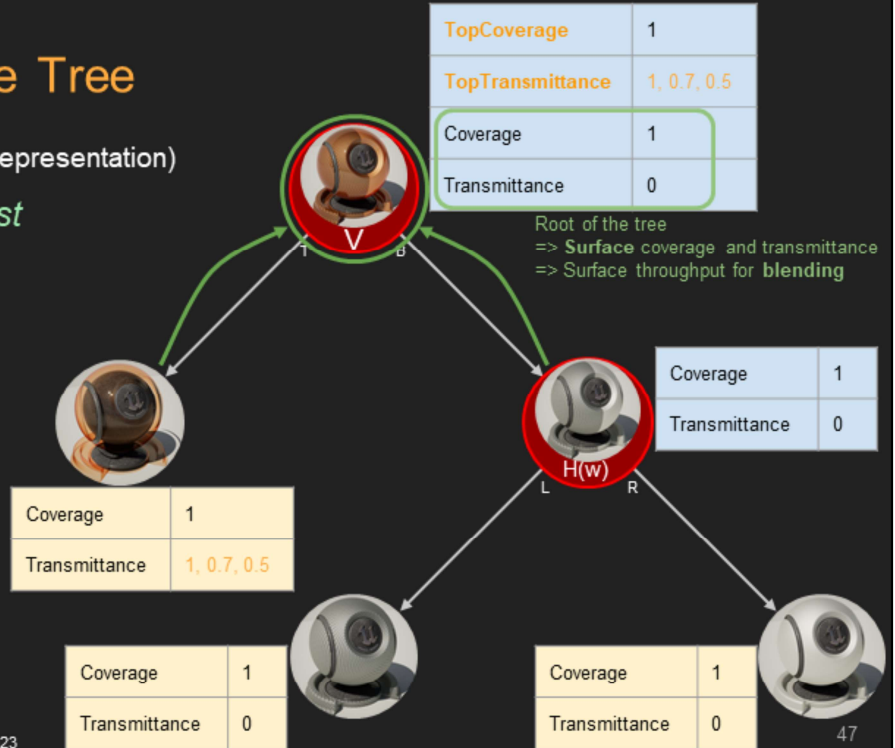
Secondly, we are going to process each operator, deeper first, in order to gather information from slabs.

So that horizontal operator here gathers a representative Coverage/Transmittance from its children. And two opaque slabs result in an opaque surface in this case.

Tree

Walking the Substrate Tree

- 1- Initialise Slabs C/T (statistical representation)
- 2- Process operators, *depth-first*



Then, the vertical operator is **the tree root: it gathers a representative Coverage/Transmittance from its children.**

Layering a clear coat on top of an opaque material still results in an opaque surface in this case.

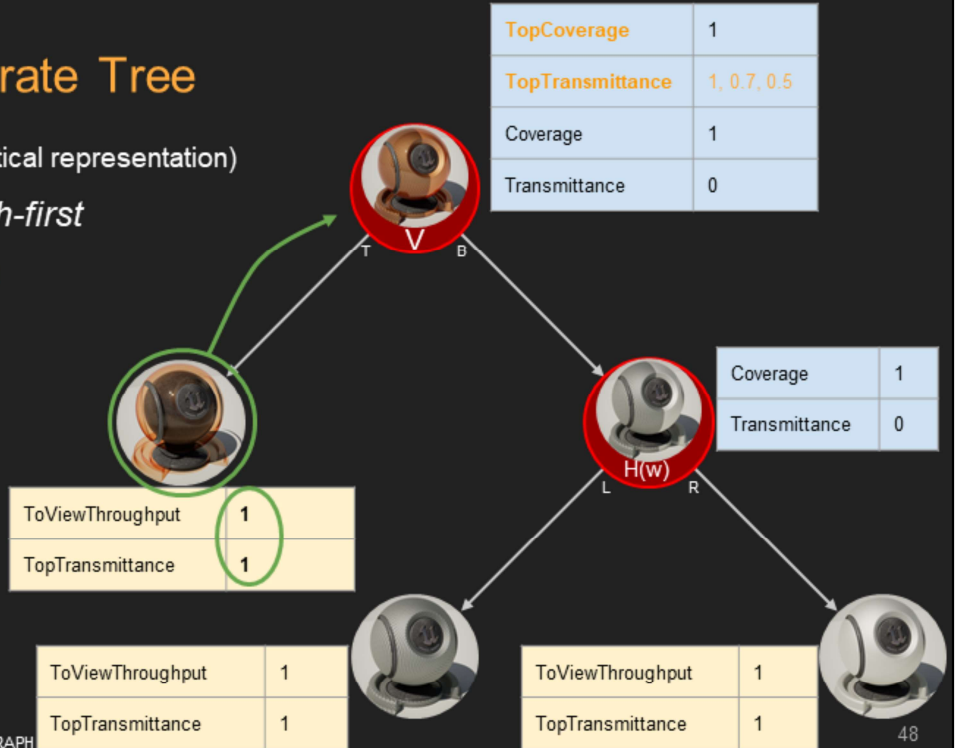
That Coverage/Transmittance value on the root node of the Substrate tree is important: it is a representation of the material total coverage and total transmittance that can be used to compute the throughput, itself used when alpha blend the surface with the scene color, if a translucent rendering mode is selected by the user.

Also, please note that we also store a separate representation of the isolated top layer Coverage/Transmittance that will be used later to compute some values.

Tree

Walking the Substrate Tree

- 1- Initialise Slabs C/T (statistical representation)
- 2- Process operators, *depth-first*
- 3- Slabs walk up the graph



Thirdly, we have each slab walk the tree up to the root node in order to compute other values such as **ToViewThrouput** or **TopTransmittance** as explained a few slides back.

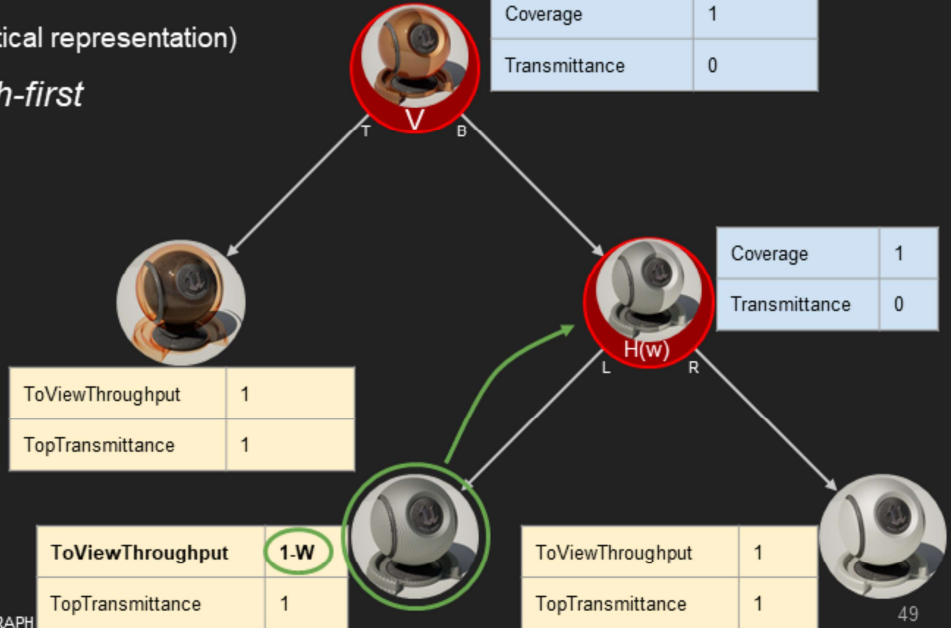
We see that the colored coat walk through the Top side of the vertical operator: its **ToViewThrouput** and **TopTransmittance** are thus unchanged.

Tree

Walking the Substrate Tree

- 1- Initialise Slabs C/T (statistical representation)
- 2- Process operators, *depth-first*
- 3- Slabs walk up the graph

TopCoverage	1
TopTransmittance	1, 0.7, 0.5
Coverage	1
Transmittance	0



The carbon fiber slab has first to walk up through the horizontal mixing operator. As such, its **ToViewThrouput** will be affected by the mixing weight according to the side it comes from.

Tree

Walking the Substrate Tree

- 1- Initialise Slabs C/T (statistical representation)
- 2- Process operators, *depth-first*
- 3- Slabs walk up the graph

```

ViewTransmittance = exp(-σ * T) // along N for a thickness T.
ViewTransmittance = exp(-σ * T * 1/NoV)
ViewTransmittance = pow(exp(-σ * T), 1/NoV)
ViewTransmittance = pow(TopTransmittance, 1/NoV)
    
```

ToViewThroughput	1
TopTransmittance	1

ToViewThroughput	$(1-W) * \text{ViewTransmittance}$
TopTransmittance	TopTransmittance

TopCoverage	1
TopTransmittance	1, 0.7, 0.5
Coverage	1
Transmittance	0

Coverage	1
Transmittance	0

ToViewThroughput	1
TopTransmittance	1

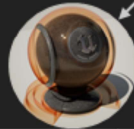
Then the carbon fiber slab walks up through the vertical layering node. It comes from the bottom side of the node, so it has its **ToViewThrouput** or **TopTransmittance** affected by **TopCoverage** and **TopTransmittance** of the vertical node.

Tree

Walking the Substrate Tree

- 1- Initialise Slabs C/T (statistical representation)
- 2- Process operators, *depth-first*
- 3- Slabs walk up the graph

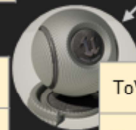
TopCoverage	1
TopTransmittance	1, 0.7, 0.5
Coverage	1
Transmittance	0



ToViewThroughput	1
TopTransmittance	1



Coverage	1
Transmittance	0



ToViewThroughput	$(1-W) * \text{ViewTransmittance}$
TopTransmittance	TopTransmittance

ToViewThroughput	$W * \text{ViewTransmittance}$
TopTransmittance	TopTransmittance

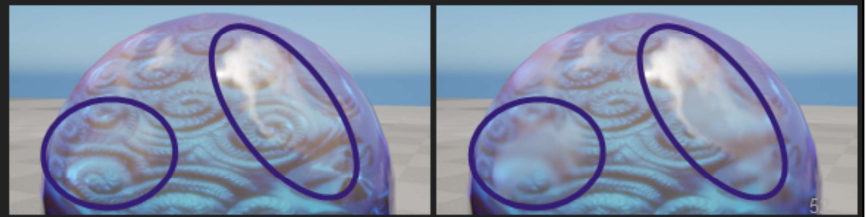
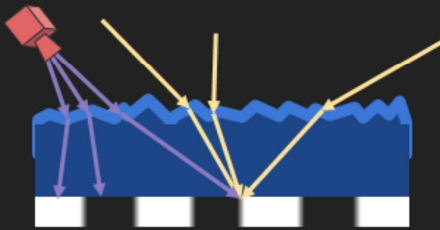


And this continues one and on for each slab.

Tree

Substrate tree walk evaluates many Closure data

- Energy preservative *Specular Transmission* from top layer
"A Multiple-Scattering Microfacet Model for Real-Time Image Based Lighting" from Carmelo J. Fdez-Aguera
- Roughness tracking: top layers roughness affects bottom specular sharpness
"Efficient Rendering of Layered Materials using an Atomic Decomposition with Statistical Operators" from Laurent Belcour
- Roughness of top layers can blur bottom layers (Opt-In)
 - Approximate specular lobe PSF as a Gaussian w.r.t. roughness and thickness (assuming air/water transition)
 - Blur in screen space separated Bottom layers \Rightarrow composite with Top layer



Advances in Real-Time Rendering in Games course, SIGGRAPH 2023

We also compute more data while processing the substrate tree such as :

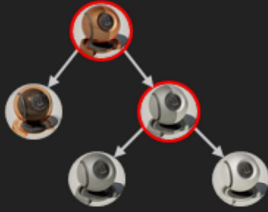
- **Energy preservative** transfert between layered slab
- We account for the **perceptual effect a rough top layer can have on a bottom layer** perceive roughness using roughness tracking. For exemple: a rough top layer will make the bottom layer slab appear rougher because the ray of incoming light will refract more at the top layer interface.
- We also account for the **refraction of view path as a function of for the top layer roughness**. The bottom layer is blurred according to the top layer roughness and thickness. The Point Spread Function of a specular lobe is approximated using a gaussian blur using an hardcoded *air to water* refraction. To achieve that, we basically separate the bottom layer and top layer into different buffers. We blur the bottom layer according to the refracted specular lobe. Then we recompose both buffers together.

Tree

Code generation for walking the tree

The translator knows about the **worst case topology**:

- Static topology \Rightarrow Static Substrate Tree
- Shared local basis (normal / tangent)
- Features



\Rightarrow Generates appropriate code for Static **tree nodes visit** in shader during base pass (helps ^{TOP} compilation)

```
----- SUBSTRATE - Default -----
StratComplInfo
- Byte Per Pixel Budget: 100
- Requested Byte Size before simplification: 76 (19 UMFT32)
- Requested Byte Size after simplification: 76 (19 UMFT32)
- Material complexity: COMPLEX
- SubstrateComplexity: 3
----- SUBSTRATE TREE - Default -----
MOP Maximum distance to leaves: 2
----- DistanceFromLeaves = 2 -----
Id=0 Op=VERTICAL ParentId=-1 LeftIndex=1 RightIndex=2 BOPId=-1 LayerDepth=0 IsTop=0 IsBot=0 BOPType=SLAB
----- DistanceFromLeaves = 1 -----
Id=2 Op=HORIZONTAL ParentId=0 LeftIndex=3 RightIndex=4 BOPId=-1 LayerDepth=1 IsTop=0 IsBot=0 BOPType=SLAB
----- DistanceFromLeaves = 0 -----
Id=1 Op=BOP ParentId=0 LeftIndex=1 RightIndex=1 BOPId=0 LayerDepth=0 IsTop=1 IsBot=0 BOPType=SLAB
Id=3 Op=BOP ParentId=2 LeftIndex=1 RightIndex=1 BOPId=1 LayerDepth=1 IsTop=0 IsBot=1 BOPType=SLAB
Id=4 Op=BOP ParentId=2 LeftIndex=1 RightIndex=1 BOPId=2 LayerDepth=1 IsTop=0 IsBot=1 BOPType=SLAB
----- SUBSTRATE - Default -----
SSS=1 WFO=1 FPO=0 Rough=0 Fuzz=1 Aniso=0 GInt=0 SpecularProfile=0
SSS=0 WFO=0 FPO=0 Rough=0 Fuzz=0 Aniso=0 GInt=0 SpecularProfile=0
SSS=0 WFO=0 FPO=0 Rough=0 Fuzz=0 Aniso=0 GInt=0 SpecularProfile=0
SSS=0 WFO=0 FPO=0 Rough=0 Fuzz=0 Aniso=0 GInt=0 SpecularProfile=0

void FSubstrateLeafShader: UpdateAllBOPOperatorCoverageTransmittanceOfSubstrateIntegralLeafSettings settings, float3 V
{
    SubstrateTree.UpdateSingleBOPOperatorCoverageTransmittance(this, 0, settings, V);
    SubstrateTree.UpdateSingleBOPOperatorCoverageTransmittance(this, 1, settings, V);
    SubstrateTree.UpdateSingleBOPOperatorCoverageTransmittance(this, 2, settings, V);
}

void FSubstrateLeafShader: UpdateAllOperatorCoverageTransmittance()
{
    // MaxDistanceFromLeaves = 1
    SubstrateTree.UpdateSingleOperatorCoverageTransmittance(0, /operator index/);
    // MaxDistanceFromLeaves = 2
    SubstrateTree.UpdateSingleOperatorCoverageTransmittance(8, /operator index/);
}

void FSubstrateLeafShader: UpdateAllBOPWithBottomOperatorVisit()
{
    SubstrateTree.UpdateAllBOPWithBottomOperatorVisit_VisualVertical(0, /BOP index/, 0, /Mip index/, 1, /PreviousInput/);
    SubstrateTree.UpdateAllBOPWithBottomOperatorVisit_Horizontal(1, /BOP index/, 2, /Mip index/, 1, /PreviousInput/);
    SubstrateTree.UpdateAllBOPWithBottomOperatorVisit_VisualVertical(2, /BOP index/, 0, /Mip index/, 2, /PreviousInput/);
    SubstrateTree.UpdateAllBOPWithBottomOperatorVisit_Horizontal(3, /BOP index/, 2, /Mip index/, 2, /PreviousInput/);
    SubstrateTree.UpdateAllBOPWithBottomOperatorVisit_VisualVertical(4, /BOP index/, 0, /Mip index/, 0, /PreviousInput/);
}
```

Some more details: **when we compile a material topology, we do it for the worst case topology**. This is needed because we need to flatten the tree parsing code. Compilers could not handle dynamic tree processing using for loop unfortunately (weird behaviors and crashes at compilation time would happen).

We also gather all the **tangent bases** and if some slabs use the same, we only store a single copy to limit allocated memory in the GBuffer.

As you can see in blue, we also track each **enabled features** for each slab in order to know more about the worst case complexity and byte per pixel requirement of the material once stored in the GBuffer. This is so that we can simplify it according to project setting driving the maximum GBuffer bytes per pixel as I will explain later.

Substrate Scalability

Controlling Complexity

54

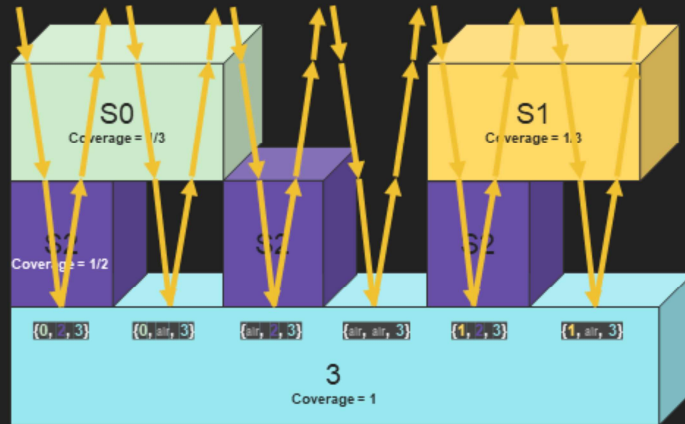
So we have seen that we can output closures from the processing of the Substrate tree made of operators and slabs.

But **how do we get that closure count under control** according to the project quality settings?

Scalability

Our solution to limit closure combinatorial explosion

Exemple with 4 BSDFs in 3 layers:



For that topology, BSDF 3 interactions are: {0, 2, 3} {0, air, 3} {air, 2, 3} {air, air, 3} {1, 2, 3} {1, air, 3}

⇒

BSDF 3 should output **6 closures**

⇒ **With our** top layer statistics,

BSDF 3 is outputting only **1 closure**.

First, we have to **fight against closure combinatorial explosion**.

We illustrate this in this example where slab 3 in blue is coverage with a complex topology of slabs in two layers, each with different coverage.

If we want to represent that material topology faithfully to its definition, we would have to account the different path light can take within this material.

For instance, light can traverse S0, then S2 then reach S3. Or it can traverse S0, then air then reach S3. And so on, leading to 6 different paths resulting in different throughput and perceived roughness for slab 3.

This should effectively results in 6 different closures that would need to be evaluated (different ToViewThroughput, TopTransmittance and specular Roughness). **That is too expensive.**

To simplify this case, we have come up with a way to statistically represent the combination of any slabs according to the operators linking them. In this case, slab 3 would then be layered with a single virtual slab aggregating the properties of all the other slabs. That would then result in a single closure.

Scalability

Our solution to limit closure count explosion

A statistical approach to represent many Slabs as one: {transmittance, coverage, roughness, refraction lobe}

Effect of Coverage Operator?



Effect of Horizontal mixing Operator?



Effect of Vertical layering Operator?



⇒ Please refer to the bonus material to know more about how we combine layers of Slabs

In this statistical representation we came up with, a **slab, or an aggregate of multiple slab, is represented by a fixed set of attributes**: transmittance, coverage, roughness and a refraction lobe.

It assumes there is no correlation between the coverage of the matter of each Slab.

For instance, what is the effect of a coverage operator on a slab? It will basically reduce the coverage of a slab, without affecting its transmittance, roughness and refraction lobe.

We also have more complex solutions for horizontal mixing and vertical layering operators. Please refer to the bonus slides for more details.

Scalability

Scaling Visual Complexity and Performance

- **Complexity control** per platform
 - PC, PS5, XB, PS4, XB1, Switch, Mobile
 - Need to reach 60hz - 120hz
- **Per platform** controls

Constraints to respect	Shading quality	Features / Effects	Must fit in memory
Forward	✓	✓	X
Deferred	✓	✓	✓

Even in forward, you might want to reduce the closure count.

Now that we have the closure count under control, we still need to scale according to a target platform and quality settings.

For instance, we can adapt

- The **shading quality**,
- The **enabled features** (roughness tracking or rough refractions)
- And, in the deferred shading case, we also have to be able to **fit the material in the GBuffer**. That is represented by an set of byte per pixel constraint. Even when forward shading is used, it could be that a project running on mobile would not want to evaluate more than a single closure per pixel.

Scalability

Tree Scalability

- How to render the following complex material with less closures?



Slab = Principled representation of matter

⇒ Our solution

- Progressive tree simplification by collapsing operators
- Using Parameter blending using empirical rules
- Not enough? Disable features (SSS, Fuzz, etc.)



This would be hard to do with a soup of BSDFs (e.g. MDL, MatX). Our principled Slab representation facilitates that approach.

So for this example of complex material with horizontally mixed and vertically layered slabs: how can we simplify the tree to reduce the amount of closure we output?

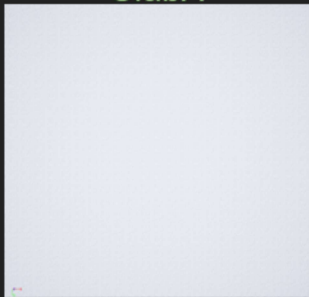
You have to remember that **a slab is a principled representation of matter**. So we understand its components and can think about ways to combine multiple slabs. Thus **our solution to the simplification problem: progressive tree simplification using parameter blend of slabs** descriptions based on some empirical rules. If that is not enough, we can even simplify by disabling special features.

It is important to note that this would not be possible to do with a soup of BSDF as proposed by MDL or MaterialX. Because a Specular BSDF and a DiffuseBSDF cannot be combined for instance.

Scalability

Tree Scalability

SlabA



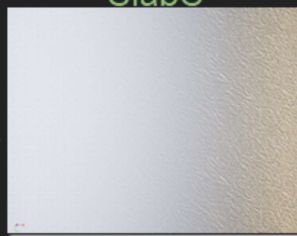
DiffuseA
F0A
F90A
NormalA

SlabB



DiffuseB
F0B
F90B
NormalB

SlabC



HorizontalMixParamBlend(SlabA, SlabB, mix) ⇒

```
lerp (DiffuseA, DiffuseB, mix)  
lerp (F0A, F0B, mix)  
lerp (F90A, F90B, mix)  
normalize (lerp (NormalA, NormalB, mix))
```

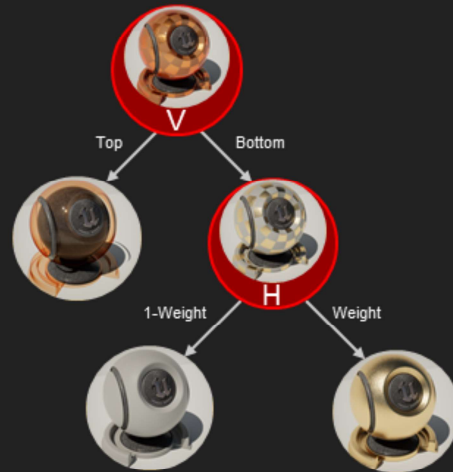
Here is a simple exemple:

- Here is a Slab A with some diffuse, F0, F90 and normal.
- And another Slab B with different diffuse, F0, F90 and normal.
- It is relatively straightforward to think that a way to parameter blend horizontal mixing is a simple lerp of the parameters.

Scalability

Tree Scalability

Collapse the Substrate tree with parameter blending

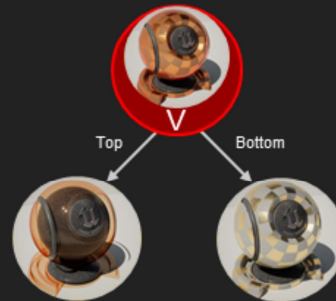


This is the default material with all the lobes, **3 closures stored as 72 bytes.**

Scalability

Tree Scalability

Collapse the Substrate tree with parameter blending

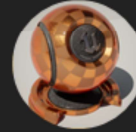
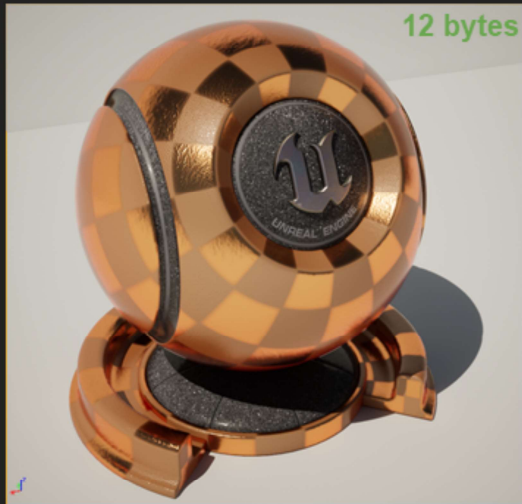


We can simplify this material by using parameter blending for the horizontal mix operator first (because we start by margin the subtree that are the deepest in the tree).
In this case, we end up with **2 slabs, resulting in 2 closures stored as 48 bytes.**

Scalability

Tree Scalability

Collapse the Substrate tree with parameter blending



Finally, we can simplify this material by using parameter blending for the vertical layering operator (because we start by margin the subtree that are the deepest in the tree).

In this case, we end up with **1 slabs, resulting in 1 closures stored as 12 bytes.**

Storage & Evaluation

Data & Closures

63

So now that we can output closures that respect the constraints for targets platforms.
Let's see how we can use them in order to light a scene.

Storage & Evaluation

Philosophy - You pay for what you use (**memory & runtime cost**)

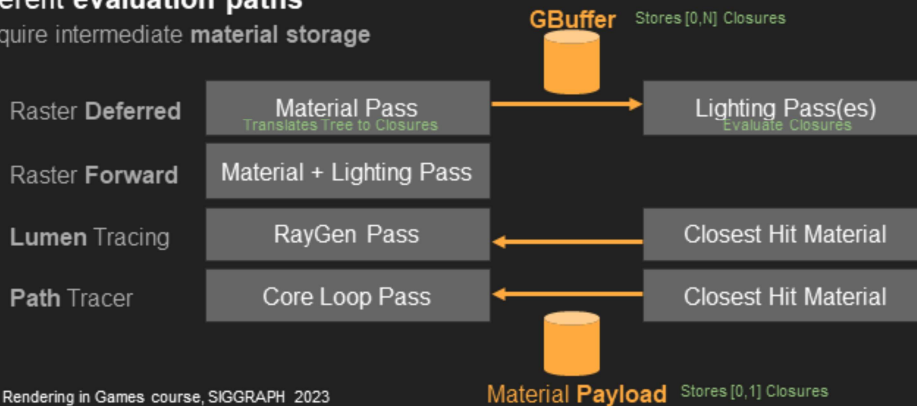
Fast standard use cases (80/20)

Per pixel storage / evaluation decisions based on used features

Support hero / advanced assets

UE has different evaluation paths

Most paths require intermediate **material storage**



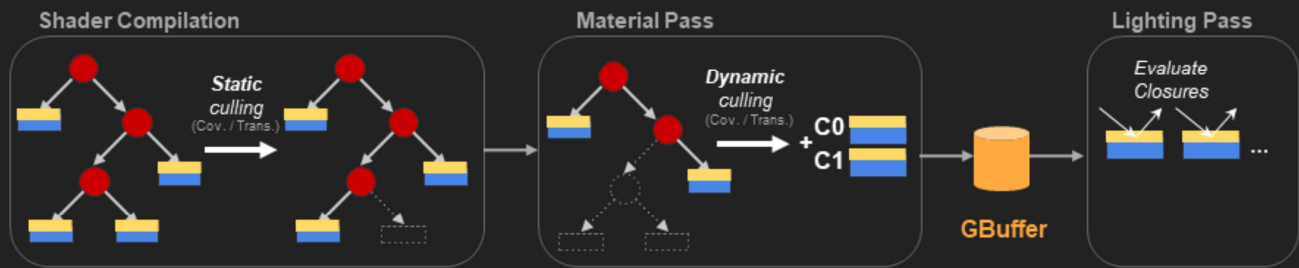
So the philosophy here is that you should only pay memory and lighting cost for what you effectively use.

We can adapt per pixel the storage and its memory footprint so the cost will adapt to what is on screen. And remains cheap if most of the materials are simple. All that while allowing more advance hero materials in some scenes.

As you may know already, unreal engine has different lighting path:

- **Deferred**, where the closures as stored in the gbuffer first and then later read by the lighting passes
- **Forward**, where lighting is evaluated in line during the base pass
- Lumen **ray tracing**, where material is stored inside a ray tracing payload
- And similarly **path tracing**.

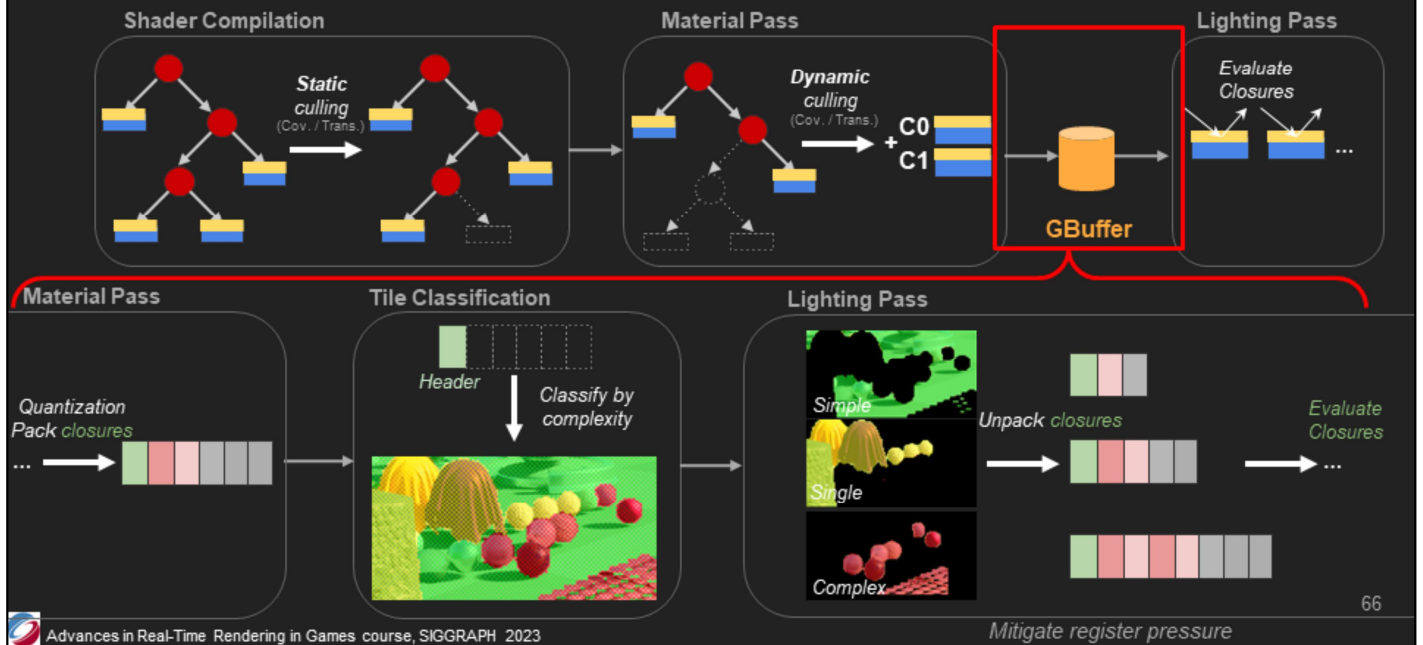
Storage & Evaluation - Deferred Shading



Let's have a look at what happens during the life of a material:

1. The **material is compiled**. If the compiler identifies that some slabs will always have a weight of 0, those will be culled away by the static compilation analysis after code has been flattened.
2. When the **base pass material shader is executed**
 - a. Closures are evaluated and written into the gbuffer.
 - b. If a closures has a weight of 0, it is skipped and removed from the list written to the gbuffer.
3. Later, **closures are read by the lighting passes** for evaluation

Storage & Evaluation - Deferred Shading



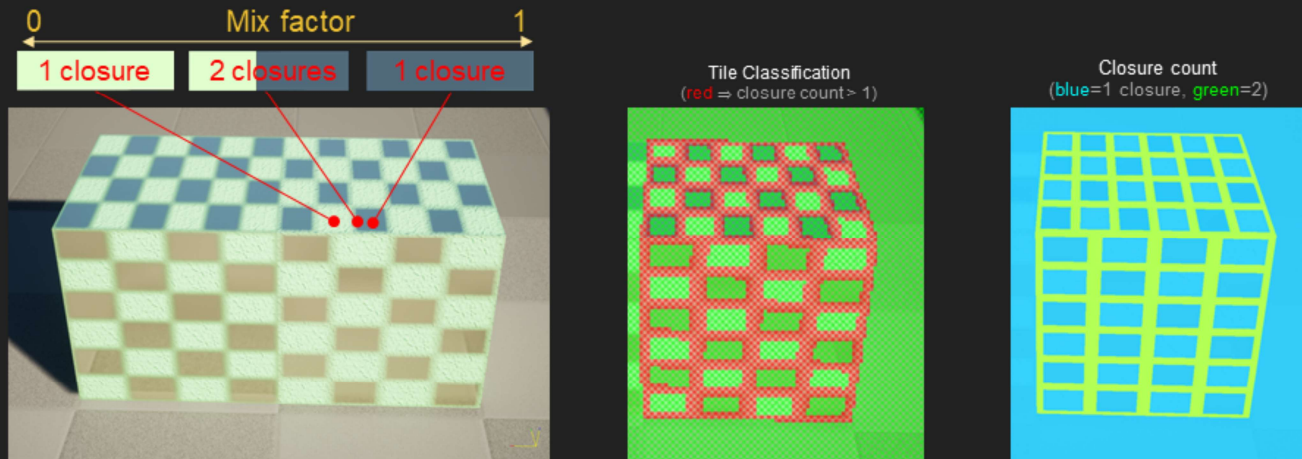
Let's have a look in a bit more details on how the gbuffer is used:

1. The material **closures** are stored in the **gbuffer with aggressive packing** to use the least amount of byte as possible.
2. The gbuffer first bits are read in order to read the complexity of each pixel and the **screen is categorised as tiles according to the required shading complexity** (simple slab, slab + features, multiple slabs, etc.).
3. **Each of our lighting passes are tiled so that the shader code can be optimised** in order to read and process the closures for lighting in an efficient way. This basically reduces register pressure, increase occupancy and reduce render cost on GPU.

Storage & Evaluation - Culling

Dynamic tree culling

Use dynamic inputs to prune tree based on Coverage & Transmittance



A small illustration of the dynamic culling of closures happening in real time (as discussed on the previous slide).

Here you can see 2 slabs been horizontally mixed together. On the tile classification debug view on the right, you can see that we only pay the cost of complex tiles where the material actually output two closures per pixel. The closure having a mixing weight of 0 is simply evicted from the evaluation passes.

Storage & Evaluation - Material Storage

Adaptive material storage based on **topology & enabled features**

You pay for what you use ⇒ simple case needs to be fast & lightweight



Packing adapted to **complexity per pixel**

- Simple** → Single closure, Diffuse / Specular / Roughness
- Single** → Single closure, any features
- Complex** → Multiple closures & features
- Special** → Expensive use cases (Glint, SpecLUT)

So what do we store per pixel in the GBuffer / material buffer?

A **header**, representing the complexity of the material for categorisation and containing closures count, tangent bases count, etc.

Then we have the **list of closures** with associated data for each closures according to the enabled features, e.g. SSS or Fuzz.

After that we have the **list of tangent bases** used by all the closures. Note that if two closures share the same basis, a single basis will be written out.

Last but not the least, we also store **other data** such as TopLayer data (e.g. normal, roughness) and SSS data in order to be able to communicate with out post processes such as SSR, SSAO, DFAO or SSS without having them reading all the material data.

That list of data is adapted according to complexity and we have different packing for different cases.

Simple



Layout Simple

Single Closure
No F90, No SSS, No Haze, etc.
No Anisotropy

Memory footprint

Total 12 bytes

NO LayerType: 1	FD Diffuse Albedo	Top Normal Top Roughness
Material Header	Closure Data	TopLayer Data
2 bytes	6 bytes	4 bytes

For instance, this is a **SIMPLE material**: only relying on diffuse, specular and roughness. Only **12 bytes of data**

Single



Topology

Slab

Layout Single

Single Closure
All features (Here: haziness)
No Anisotropy

Memory footprint

Total 20 bytes

Material Header	Closure Header	Closure Data	TopLayer Data
2 bytes	2 bytes	12 bytes	4 bytes

This is a **SINGLE material**: a slab with any features. In this case, haziness is used. Number of bytes per pixel used goes **up to 20 bytes per pixel**.

Complex



Topology

○ ○ ○ ○ ○	○ ○ ○ ○ ○
Slab	
○ ○ ○ ○ ○	○ ○ ○ ○ ○
Slab	Slab

Layout Complex

Multiple Closures
All features
Anisotropy

Memory footprint

Total 76 bytes

ID LayoutType: 0
 BSSDF Count: 2
 Tangent Basis Count: 1
 Tangent Basis 0: Normal

Shared tangent basis between layers		
Material Header	Tangent basis 0	Tangent basis 1
4 bytes	4 bytes	4 bytes

64 bytes	Closure Header	Closure Data
	Closure Header	Closure Data
	Closure Header	Closure Data

Tangent Basis ID: 0
 Is Top Layer: 1
 Has Anisotropy: 0
 ...

FO
 Diffuse Albedo
 Roughness
 MIP

This is a **COMPLEX material**, feature many slabs.

You may recognise the Opal material feature in our latest GDC demo: Electric Dream. It consists of two raymarched height field slabs with colored specular layered with a clear coat.

This is stored using a general representation of a material as closures. **It uses 76 bytes of data per pixel.**

Special

Layout Special

Multiple Closures
All features
Glints
Specular LUT

Memory footprint

Total 100 bytes

Material Header	Tangent basis
4 bytes	4 bytes
92 bytes	
Closure Header	Closure Data
Closure Header	Closure Data
Closure Header	Closure Data
Closure Header	Closure Data

NO LayoutType: 0
BSPF Count: 2
Target Basis Count: 1
Target Basis ID: Normal

Target Basis ID: 0
Is Top Layer: 1
Has Anisotropy: 0

FD
Diffuse Albedo
Roughness *MIP* ...

This is a **SPECIAL material**, featuring advanced visual effects such as Glint or SpecularLUT.
 This can be used for instance to accurately render realistic car paint with mesoscopic faceted glint effect or the advanced pearlescent look of triple layer paints.

Storage & Evaluation - Storage

Quantize closures parameters with dithering

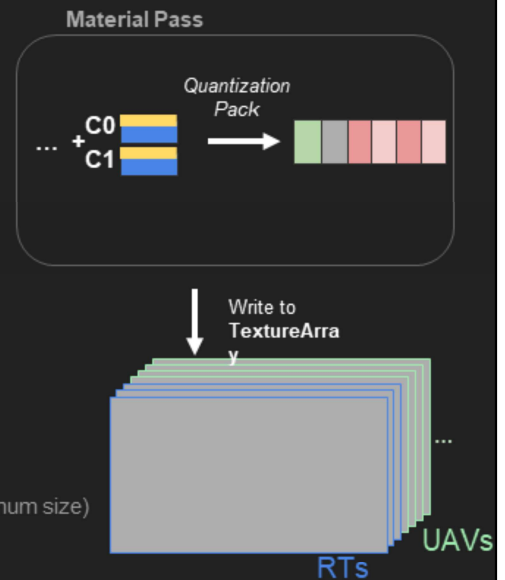
- F0 → RGB 20bit (7/7/6bits)
- DiffuseAlbedo → sRGB 20bit (7/7/6bits)
- FuzzAmount → 6bits
- ...
- Packed data into a **stream of UINTs**

Outputs closures to a Texture Array

- Layers 0-2 → **RTs**
- Layers 3+ → **UAVs**
- Lower platforms leverage simplification and avoid UAV write

Preferred layer allocation strategy

- Grow w.r.t. the most demanding visible materials (up to the project maximum size)
- Do not shrink (avoid reallocation hitches & memory fragmentation)



For each closure, their parameters are aggressively packed using quantization. And we use dithering in order to avoid seeing any banding artefact. All the closures in the end form a stream of UINT that is written into our GBuffer.

Our **GBuffer is a 2D texture array** of UINT (we like to, we like to call it material buffer).

During the base pass, the **first 3 layers are mapped to Render Target Output**, because it is faster on some hardware to leverage the caching mechanism of the merger output.

The **remaining layers are mapped onto a single UAV** and all the UINT overflowing the RT count simply are written through it.

On low end platforms, our goal is to avoid more expensive UAV writes by packing all our legacy material to 3 UINT maximum.

When it comes to the allocation strategy of that buffer: we **allow it to grow** its slice count according to what material is present on screen. But we **never shrink** it back in order to avoid memory re-allocation hitches and fragmentation.

Storage & Evaluation - Lighting evaluation

Lighting evaluation based on **enabled features**

- Multiple Closures evaluated as a **Loop**

```
void Evaluate()
{
    Substrate_For(uint ClosureIndex = 0; ClosureIndex < SubstrateHeader.ClosureCount; ClosureIndex++)
    {
        // ...
    }
}
```

- **Dynamic branching** for each features

```
void EvaluateClosure()
{
    // ...
    if (CLOSURE_GETHASHAZINESS(BSDFContext))
    {
        // ...
    }
    if (CLOSURE_GETHASHFUZZ(BSDFContext))
    {
        // ...
    }
    if (CLOSURE_GETSSSType(BSDFContext) == SSS_TYPE_DIFFUSION)
    {
        // ...
    }
    // ...
}
```

Lighting Pass



Classification + lighting permutations

- Mitigate register pressures
- Ensure standard case remains fast

Once the closures are packed and stored in the GBuffer, we can read them back for lighting evaluation.

We simply **loop over all closures**, we load them. For each of this closure we can **evaluate the lighting according to the enable features**.

And as mentioned before, the tile complexity classification helps optimise the GPU cost of each of those passes.

Storage & Evaluation - Ray / Path Tracing

Hardware Ray Tracing

- Performance \Rightarrow Payload size < 64 bytes


Ray Tracing with Lumen

- Material lighting accuracy not critical for GI
- Use **Tree full simplification**
 - single closure fit in 64 bytes

Path Tracer

- On a hit **Stochastically** select a Closure
 - Based on **Directional Albedo**
 - Avoid storing **entire** topology and Closures
- A Tradeoff
 - Variance versus performance
 - Typically fine: lots of samples + denoising



 Advances in Real-Time Rendering in Games course, SIGGRAPH 2023

When it comes to hardware ray tracing, as you may know, we need to send the material data from the hit shaders to the RGS (Ray Generation Shader). **This is done through the ray tracing payload structure. And this structure needs to remain small in order to be efficient.** We restrict ourselves to 64 bytes.

However, **we cannot send the entire substrate tree** material for processing into the RGS because it simply would not fit when multiple slabs are present. Similarly, multiple closures would not fit in many cases.

For **Lumen**, were all the lobes and reflections details are less important for global illumination, we **enforce a full simplification of the substrate tree** to a single slab which will output a single closure. A single closure can thus fit in the payload and then be evaluated by lumen in the RGS.

However, for the **path tracer**, we are interested in all the reflection details and interactions. In this case, we simply **stochastically select a closure for the list resulting from the substrate tree processing in the hit shader based on its directional albedo**. That closure pdf is then reweighted accordingly to all the closure directional albedo. Then the path tracer can continue its job of selecting and sample a lobe of that closure.

That is a choice, trading off increased variance for better performance. With a little bit more samples and our denoiser, we are able to reach noise free result in the use cases we have tested. As you can see on those images.

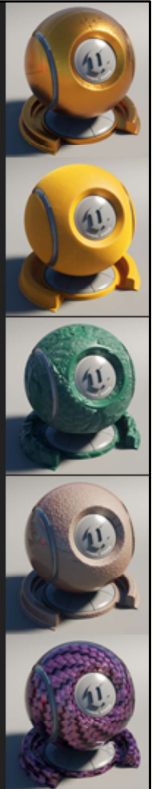
Storage & Evaluation - Performance

		Legacy		Substrate	
		Material	Lighting	Material	Lighting
Simple <i>VGPRs 40</i>	Metal	0.08	0.20	0.10	0.20
Single <i>VGPRs 56</i>	Cloth	0.09	0.21	0.12	0.25
	SSS	0.09	0.22	0.15	0.24
Complex <i>VGPRs 128</i>	2 layers			0.23	0.58
	3 layers			0.31	0.79

*Cloth models are different
Cost due to extra-SSS data*

+0.02 / +0.06 +0.02 / +0.04

- Timings in ms for a fullscreen material on PS5 @ 1080p
- Classification mitigates occupancy
- Tile classification: 0.03ms (0.18ms async)
- Dispatch complex tiles first to avoid long tail



Here, we give deferred shading performance for the GBuffer lay down and light passes on PS5 @1080p for a full screen material and lighting.

You can see that

- For the **simple case**, we are on par,
- For the **single case** we do have a small overhead related to the extra buffers we have to write in order to communicate with our post process passes through the TopLayer and SSS data buffer. This is something we want to improve going forward.
- For the **complex case**, not supported by the legacy GBuffer, the cost will increase linearly with the number of closure as expected.

Storage & Evaluation - Memory @1080p

Legacy

GBufferA (normal/obj data)	8.0	MB	// all GBuffer textures are RGBA8
GBufferB (Met/Spec/Rough/ShadId)	8.0	MB	
GBufferC (BaseColor/Occlusion)	8.0	MB	
GBufferD (CustomData)	8.0	MB	
TOTAL	32.0	MB	

Substrate

Substrate.Material	50.0	MB	// 4 UINT for SINGLE Closure + 2 SSS data => 6 UINT slices
Substrate.TopLayerTexture	8.0	MB	// Roughness / Normal
TOTAL	58.0	MB	// (Legacy + 24MB)

When it comes to memory allocations, you can see here the legacy gbuffer memory usage.

When substrate is enabled, we do allocate more memory. This overhead is related to the extra buffers we have to write in order to communicate with our post process passes through the TopLayer and SSS data buffer. This is something we want to improve going forward.

Storage & Evaluation - Memory @1080p

Remember: not all bytes are written!

SIMPLE	Legacy Disney Diff/Spec/Rough	12 bytes	// 2 slices + TopLayerData
SINGLE	Legacy shading models	16 bytes	// 3 slices + TopLayerData
SINGLE	previous + 2ndRoughness + F90	20 bytes	// 4 slices + TopLayerData
SINGLE	previous + Cloth	24 bytes	// 5 slices + TopLayerData
COMPLEX	previous + Anisotropic Spec	28 bytes	// 6 slices + TopLayerData
COMPLEX	previous + 1 Coating Slab	40 bytes	// 9 slices + TopLayerData
COMPLEX SPECIAL	previous + SpecLUT	44 bytes	//10 slices + TopLayerData
COMPLEX SPECIAL	previous + Glints	60 bytes	//14 slices + TopLayerData

More

- Optimised 12 bytes path for legacy shading models
- Only write though 3 RTs (avoid slower UAVs write on older platforms)

⇒ **And we are not done yet!** (optimisation/performance work is in progress...)

However, it is very important to remember that even if we allocate more memory, we do not write it all. We adapt the memory footprint with the complexity of the material per pixel.

We have also some special optimised paths for some of the legacy shading models we want to precisely maintain the cost going forward.

And we are not done yet, optimisation for memory and performance are going to be worked on from now on.

Conclusion

Conclusion

Modern material framework built around **Slabs**, **Operators**, and **Tree**

- Scale from **simple** to **hero** materials
- Built-in material simplification system for scalability
- Extensibility for advanced use cases (Glint, SpecularLUT, ...)
- Unleash artists creativity: from **Fixed Shading Models** to **Composition framework** (Be aware of the complexity and cost!)

Support

- Support most of UE systems (Raster / Lumen / RT / Path-tracer)
- Support most platforms (Curr-Gen Console, Prev-Gen Console, Switch, mobiles ...)
- Fast path for games

Timeline

- Shipped on *Electric Dreams* Demo
- Focus on production-readiness - Performance and memory

To conclude, we have presented **Unreal Engine Substrate**: what we believe is a **modern material framework that will scale the performance and cost of material from hero to simple one**. Together with a scalability system to adapt to many platform Unreal has to support.

This is our solution to a scalable GBuffer, something we believe is crucial to be able to move forward in the future and be able to scale the visual complexity of material in the future.

We are able to also include advanced use cases such as glints or specular lut, without impacting the cost of simpler materials.

This system **unleash artist creativity**, freeing them from the fixed shading model list we had before. **Opening the door to a composable material framework**.

Substrate is **supported in most of our systems and platforms**, with scalability controls. Maintain critical performance for platform that requires it, scaling quality with refresh rates or quality as priority.

When it comes to the timeline, we have shipped the electric dreams demo during GDC 2023 with Substrate enabled across the board.

Next, we are going to focus on production readiness of Substrate, with more performance and memory optimisations.

Current limitations

Requirements

- **Depth prepass + EarlyDepthTest** is mandatory
 - large material outputting to UAV
- **Separated Decal pass (DBuffer)** is mandatory
 - *Gbuffer is not blendable anymore*

Shader compilation challenges

- Shader **size** due to many path
- Shader **compilation time** (have to unroll Tree parsing)

Others

- Requires extra buffers: Normal/roughness for SSR, SSAO, DFAO
- Lots of UX experimentation needed: **steep learning curve for some artists!**



There are **some limitations** you have to be aware of.

First, we **require a depth prepass**. This is due to the possibility of some of the material writing material and closures data through UAV.

Secondly, we **require a separated decal accumulation pass**, often called DBuffer. This is needed because the GBuffer is not blendable anymore. So decals needs to be composited after, during the base pass over the closures that are generated.

There are also some **challenges related to shader compilation**: more compilation and packing path that can be encountered during the base pass. So shaders are bigger and also takes more time to compile. Nothing comes for free but this is a situation we want to improve upon.

We would also like to mitigate the extra cost & overhead related to the extra buffers we have to write in order to communicate with our post process passes through the TopLayer and SSS data buffer.

Last but not least, we would like to **experiment with more user experience improvements** because the slab parameterisation in itself has a steep learning curve as compared to the previous parameterisation we had. This is likely going to come with a way to abstract that complexity with simpler parameterisation, for instance mapping to the shading model we have before, that artists are used to.

Future

Better energy transfer in between layers

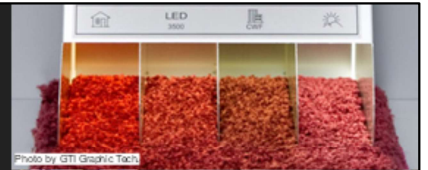
Taking into account relative IOR & bounces

Spectral Integration

Accurate color rendition, metamerism, natural reflectance, fluorescence, dispersion...

Optically thin volumetric materials

Phase function
Single & Multiple scattering
Transition from optically thick (SSS) to thin/translucent
Self Shadow
Colored shadow and Heterogeneous internal representation
Support deformable/animated meshes
Consider inner parts (bones)
Opalescence



Substrate open the box to many more improvements down the line:

- We would like to have a **solution for the multiple refraction and reflection bounces** that can happen in between multi layered materials
- We would like to **experiment with spectral rendering** in order to simulate metamerism of fluorescence for instance
- Last but not least, we would like to **extend this framework to support material with volumetric content**, rendering colored multiple scattering effects, self shading, internal details such as bones and organs, etc.

The end

Thanks to

- Natasha Tatarchuk
- Epic Games and the Unreal Engine rendering team
- *Electric Dream* demo team

Any questions?

Thanks you for reading our presentation notes!
Do not hesitate to contact us for any interesting discussions.

Bibliography

- Physically Based Shading at Disney*, Brent Burley, SIGGRAPH 2012 Course: Physically Based Shading in Theory and Practice.
- Real Shading in Unreal Engine 4*, Brian Karis, SIGGRAPH 2013 Course: Physically Based Shading in Theory and Practice.
- Moving Frostbite to PBR*, Sébastien Lagarde & Charles de Rousiers, SIGGRAPH 2014 Course: Physically Based Shading in Theory and Practice.
- Extending the Disney BRDF to a BSDF with Integrated Subsurface Scattering*, Brent Burley, SIGGRAPH 2015 Course: Physically Based Shading in Theory and Practice.
- Practical Multilayered Materials in Call of Duty: Infinite Warfare*, Michal Drobot, SIGGRAPH 2017 Course: Physically Based Shading in Theory and Practice.
- Revisiting Physically Based Shading at Imageworks*, Chris Kulla & Alejandro Conty, SIGGRAPH 2017 Course: Physically Based Shading in Theory and Practice.
- A Composite BRDF Model for Hazy Gloss*, Barla et al., Computer Graphics Forum 2017.
- Material Advances in Call of Duty: WWII*, Danny Chan, SIGGRAPH 2017 Course: Advances in Real-Time Rendering in 3D Graphics and Games Course.
- Efficient Rendering of Layered Materials using an Atomic Decomposition with Statistical Operators*, Laurent Belcour, SIGGRAPH 2018.
- Practical multiple scattering compensation for microfacet models*, Emmanuel Turquin, 2019.
- A Multiple-Scattering Microfacet Model for Real-Time Image Based Lighting*, Carmelo J. Fdez-Aguera, Journal of Computer Graphics Techniques 2019.
- Real-time subsurface scattering with single pass variance-guided adaptive importance sampling*, TianTian Xie et al., i3D 2020
- Real-Time Geometric Glint Anti-Aliasing with Normal Map Filtering*, Xavier Chermain et al., i3D 2021.
- Practical Multiple-Scattering Sheen Using Linearly Transformed Cosines*, Tizian Zeltner et al., SIGGRAPH 2022.
- PBRT, Matt Pharr et al., [web page](#)
- Real-Time Rendering 4th Edition*, Tomas Akenine-Möller et al., [web page](#)



Bibliography for all the papers and book that have been extensively used and implemented along the development of Substrate for Unreal Engine.

Bonus slides



Here are some bonus slides coverage in more details some of the parts that were left off during the presentation for the sake of time.

Slab participating media

Mentioned on [slide 29](#).

When a slab is optically thin (relatively large mean free path), we have to be able to represent its medium as a participating medium with absorption and scattering.

Slabs

Defining matter: Slab Medium

Custom LUT-Based scattering

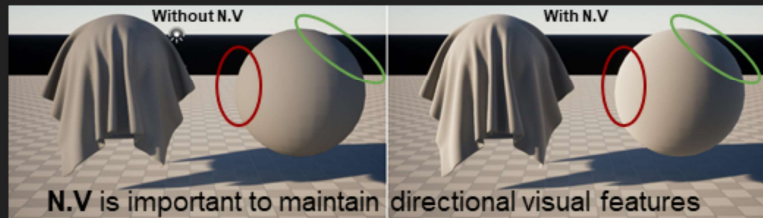
Material inputs

- **DiffuseAlbedo** is *post* multiple scattering assuming infinitely thick slab
- **MFP** (expressed for a thickness of 1 meter)

Shading

- **Output Transmittance** (analytical)
- **Output Scattered luminance** as a function of **Single Scattering Albedo**, **MFP**, **N.L** and **N.V**
 - Recover **Single Scattering Albedo from BaseColor** *"Revisiting Physically Based Shading at Imageworks"* From Kulla & Conty 2017
 - Use Precomputed **4D LUTs** storing luminance transfer for a unit light

Volumetric effect approx. as a BSDF → Missing large scattering causing hazy look (for future work)



In the future we want to use an analytical solution instead of the 4D LUT that is expensive to sample for each RGB component.

Merging Slabs

Mentioned on [slide 57](#).

How do we affect/merge slabs according to operators?

Remember: this is used to represent a single slab to affect slabs below after a layering operation

Tree

Coverage weight node update



Given a sub tree:

- A with coverage CI and transmittance TI

$$C = w CI \Rightarrow \text{New coverage}$$

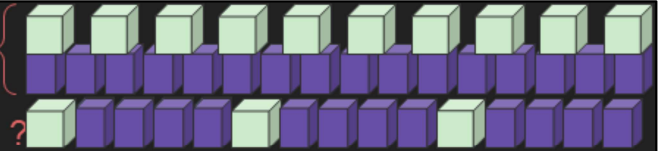
$$T = TI \Rightarrow \text{Propagate unchanged transmittance}$$



This statistical representation assumes there is no correlation between the coverage are by the matter of each Slab.

Tree

Horizontal mixing node update



Given two sub tree:

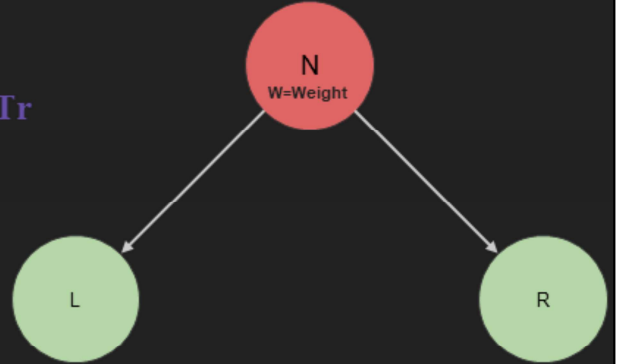
- Left with coverage C_l and transmittance T_l
- Right with coverage C_r and transmittance T_r

$$C = (1-w) C_l + w C_r$$

⇒ Weighted sum of coverages

$$T = ((1-w) C_l T_l + w C_r T_r) / \max(\epsilon, C)$$

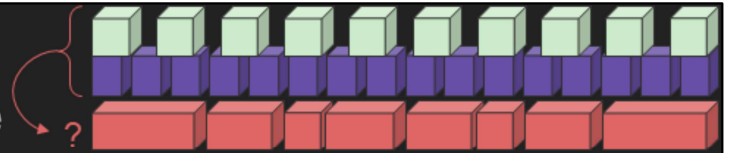
⇒ Weighted sum of transmittances rescaled by coverage to get the transmittance of medium itself



This statistical representation assumes there is no correlation between the coverage are by the matter of each Slab.

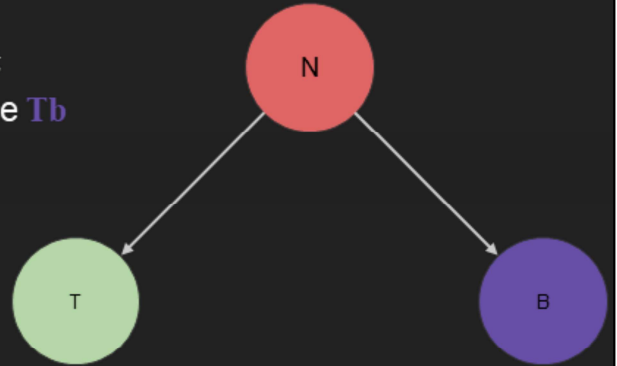
Tree

Vertical layering node update



Given two sub tree:

- Top with coverage C_t and transmittance T_t
- Bottom with coverage C_b and transmittance T_b



What is the resulting C and T ?

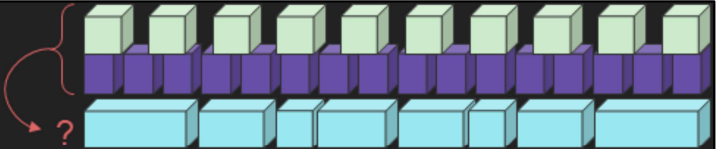
What is the portion of visible surface of Bottom only?

What are the statistics here?

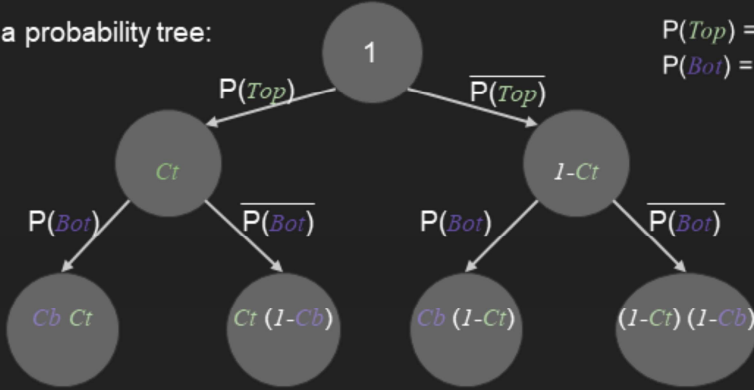
This statistical representation assumes there is no correlation between the coverage are by the matter of each Slab.

Tree

Vertical layering node update



Use a probability tree:



$P(Top) = Ct \Rightarrow$ probability to hit the Top surface
 $P(Bot) = Cb \Rightarrow$ probability to hit the Bottom surface

$$C = Cb Ct + Ct (1-Cb) + Cb (1-Ct) \Rightarrow C = Ct + Cb (1-Ct)$$

$$T = Tt (Ct (1-Cb)) + Tb (Cb (1-Ct)) + Tt Tb (Cb Ct)$$

This statistical representation assumes there is no correlation between the coverage are by the matter of each Slab.

Tools

Some details about tools and visualization we have.

Tooling & helper in editor

- **Help casual users** - Advanced concept are not easy to grasp (MFP / F90 / Thickness / ...)
 - Simple abstracted nodes (Disney, Cloth, Clear Coat, ...)
 - Helper nodes (transmittance to MFP, ...)
 - Substrate tree compilation details

- ▼ **Substrate** Building Blocks
 - Substrate Coated Layer
 - Substrate Standard Surface Opaque
 - Substrate Standard Surface Translucent
 - Substrate UE4 Default Shading
 - Substrate UE4 Unlit Shading
- ▼ **Substrate** Helpers
 - Substrate FlipFlop
 - Substrate Haziness-To-Secondary-Roughn
 - Substrate IOR-To-F0
 - Substrate Metalness-To-DiffuseColorF0
 - Substrate Rotation-To-Tangent
 - Substrate Thin-Film
 - Substrate Transmittance-To-MeanFreePat
 - Substrate View-Dependent-Coverage

Material simplification preview

Full simplification Override bytes per pixel 80 Apply to preview

Material topology preview

BSDF (SSS MFP)
BSDF (Ani)

Material advanced details

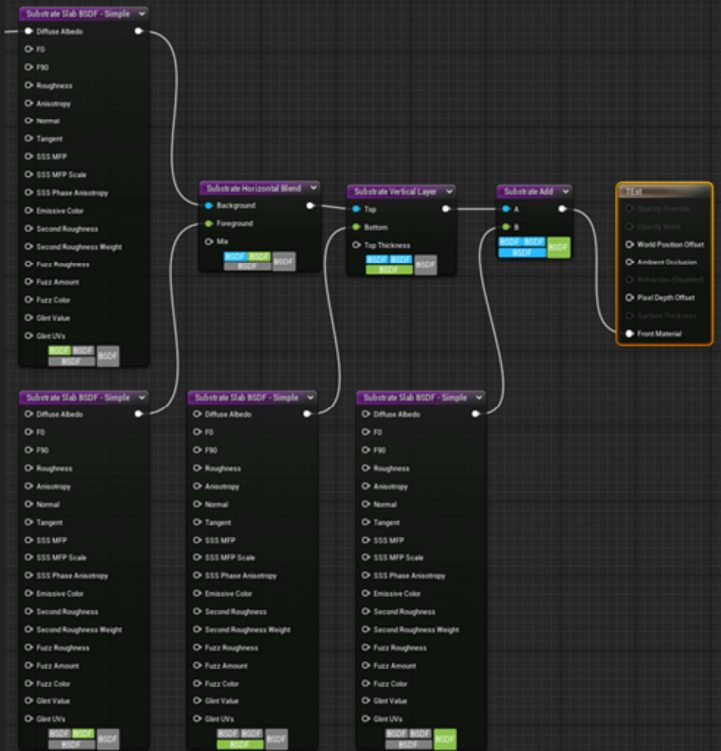
Material per pixel byte count= 48 / budget = 80
BSDF Count = 2
Local bases Count = 2
Material complexity = COMPLEX
Root Node Is Thin = 0
Material Type = Multiple Slabs

-----Detailed Output-----

```
- SharedLocalBasesCount 2
----- SUBSTRATE - Default -----
StrataCompilationInfo =
- Byte Per Pixel Budget 80
- Requested Byte Size before simplification 48 (12 UINT32)
- Requested Byte Size after simplification 48 (12 UINT32)
- Material complexity COMPLEX
- TotalBSDFCount 2
  - SLAB - SharedLocalBasisIndexMacro = SHAREDLOCALBASIS_INDEX_0_0
  - SLAB - SharedLocalBasisIndexMacro = SHAREDLOCALBASIS_INDEX_1_0
```


Tooling & helper in editor

- **Help** users with per node visualization of
 - Topology
 - Slab position within topology
 - Complexity



Substrate tree advanced visualization

The image displays a complex interface for visualizing a substrate tree. On the left, a console window shows a series of log entries for operators, including parameters like `OperatorIndex`, `FacetIndex`, `LayerDepth`, `MaxDistanceFromLeaves`, `Type`, `RightIndex`, `Height`, and `Coverage`. The main area is divided into several sections: a grid of material preview spheres and cubes, a tree diagram with nodes labeled `00`, `01`, and `02`, and a 3D scene showing a red sphere with a circular opening. A text box at the bottom center contains the text "Debug pixel pointed by mouse in editor" with a white arrow pointing to a specific location on the red sphere.