

Lux

Please note: Lux as well as this documentation are still WIP.

In case you use Lux Personal please make sure you check for the latest updates:

Unity 5.6.

<https://github.com/larsbertram69/Lux-2.02-Personal>

Unity 5.3. – 5.5.

<https://github.com/larsbertram69/Lux-2.01>

Always make sure you find the latest version of the [documentation](#).

Changelog

16-May-2017 Lux 2.02 for Unity 5.6. published.

01-Apr-2017 Lux AutoLight updated to support Unity 5.6. Get the package from [here](#).

02-Nov-2016 Lux Plus updated

18-May-2016 Thin layer wetness added, small dx9 fix

Table of Content

[Lux](#)

[Overview](#)

[Lux vs. Lux Plus](#)

[Lighting](#)

[Area Lights and diffuse fill lights](#)

[Lux BRDFs and further lighting features](#)

[Further notes about anisotropic lighting](#)

[Deferred anisotropic lighting](#)

[Specular Anti Aliasing \(Lux 2.02\)](#)

[Horizon Occlusion \(Lux 2.02\)](#)

[Surface Features](#)

[Dynamic Weather](#)

[Snow](#)

[Wetness and Water](#)

[Mix Mapping](#)

[Parallax Mapping and Parallax Occlusion Mapping](#)

[Using Lux](#)

[Setting up your Project](#)

[Forward rendering and fog](#)

[Lux Plus and Image Effects](#)

[Cinematic Image Effects](#)

[Post Processing Stack](#)

[Amplify Occlusion](#)

[Other Image Effects](#)

[Setting up your Scenes](#)

[Using the Lux standard shader](#)

[Rendering Mode](#)

[Culling](#)

[Lighting](#)

[Standard Lighting](#)

[Translucent Lighting](#)

[Anisotropic Lighting](#)

[Mix Mapping](#)

[Diffuse Scattering](#)

[Parallax Occlusion Mapping](#)

[Dynamic Weather](#)

[Lux Setup Script](#)

[Translucent Lighting](#)

[Skin Lighting](#)

[Anisotropic translucent Lighting](#)

[Lux Dynamic Weather Script](#)

[Using diffuse fill lights](#)

[Using Area Lights](#)

[Lux Area Light Script](#)

[Area Lights and Shadows](#)

[Using the Lux TexturePostprocessor](#)

[Writing custom Surface Shaders](#)

[Introduction](#)

[Adding support for Area Lights](#)

[Adding Lux surface features](#)

[Shader Properties](#)

[The Input Structure](#)

[The Vertex Function](#)

[The Surface Function](#)

[Initialize the Lux fragment structure](#)

[Initialize Dynamic Weather](#)

[Do your regular stuff](#)

[Convert metallic to specular output](#)

[Finally apply dynamic weather](#)

[The Keywords](#)

[Adding Mix Mapping](#)

[Mix mapping and PM/POM](#)

[Adding Parallax or Parallax Occlusion Mapping](#)

[Parallax or Parallax Occlusion Mapping and Alpha Testing](#)

[Adding Diffuse Scattering](#)

[Adding Specular Anti Aliasing](#)

[Adding Tessellation](#)

[Hierarchy within the surface function](#)

[Using Lux Lighting features and functions](#)

[Area lights](#)

[Diffuse fill lights](#)

[Lambert lighting](#)

[Anisotropic lighting](#)

[Advanced anisotropic lighting](#)

[Skin shading](#)

[General Texture Import Settings](#)

[Custom example Surface Shaders](#)

[Lux custom Standard Shaders](#)

[Terrain Shader](#)

[Standard WavingGrass](#)

[Simple Metallic](#)

[FullFeatures CustomSnowMask](#)

[FullFeatures DoubleSided Cutout](#)

[Tessellation DynamicWeather](#)

[Tessellation DynamicWeather MixMapping](#)

[Standard Refraction Geometry](#)

[Lux custom Translucent Shaders](#)

[Translucent Base](#)

[Translucent DynamicWeather](#)

[Translucent WavingGrass](#)

[Lux custom Anisotropic Shaders](#)

[Anisotropic Base](#)

[Anisotropic Hair](#)

[Hair shader specific Properties](#)

[Hair specific Lighting](#)

[Lux custom deferred decal shaders](#)

[Standard DeferredDecal Parallax](#)

[Standard DeferredDecal FullFeatures](#)

Overview

Lux is a (mostly...) open source shader framework built upon unity's rendering and shading pipeline. It adds advanced lighting features such as area lights, translucent and skin lighting and allows you to easily use effects like dynamic weather, mix mapping or parallax occlusion mapping.

Lux has been built having deferred rendering in mind. So you will have to face some limitations as far as you material definitions are concerned as lux aims to be efficient rather than make everything possible.

Lux ships with a standard shader which will allow you to use and adjust most features simply using the material editor. However getting most out of it you may consider writing your own surface shaders. In order to make this as easy as possible the framework ships with a bunch of includes and shader macro definitions which should do most of the work that is needed to make everything work.

Lux vs. Lux Plus

Lux ships in two different packages:

1. A free and open source one which contains all features including translucent, skin and anisotropic lighting. Latter however will always render using forward.
2. Lux Plus which is sold for a small and affordable amount on the asset store supports translucent, skin and anisotropic lighting completely rendered using deferred shading.

Lux and Lux Plus have been successfully tested on OpenGLCore, DX11 and DX9 using nvidia GPUs.

Lighting

Area Lights and diffuse fill lights

Lux supports tube, sphere (both based on point lights) and disc (spot lights only) **area lights** which can be globally enabled and disabled. When using forward rendering area lights might even be enabled on a per material basis. [Read more >](#)

Diffuse fill lights allow you to add point or spot lights which mainly affect diffuse lighting while specular highlights might be completely suppressed or simply dimmed in order to create “traditional” fill lights and fake global illumination. [Read more >](#)

Lux BRDFs and further lighting features

Lux supports unity’s built in physically based **standard lighting** which is pretty much used as is (GGX BRDF) — except for the fact that Lux adds “**Lambert**” lighting to the deferred shading path: If the specular color is set to 0 (which is outside the range of physically based shading) no specular highlights from direct lighting will be rendered: Useful e.g. for grass which can be rendered deferred without giving you strange specular reflections at grazing angles (grass shader included). Please note: In fact the shader only tests if red == 0.0.

Translucent lighting gets added on top of standard lighting. It is based on a well known solution presented by DICE and was first brought to Unity by Farfarer. It is not physically correct but still delivers nice and believable results as far as deep subsurface scattering is concerned while being pretty fast to render.

In order to not loose translucent lighting on self shadowed objects Lux allows you to suppress shadows on translucent objects.

If the “**Scattering Power**” is set to 0.0 the shader uses wrapped NdotL based translucent lighting – suitable for single sided geometry and foliage rendering (Forward Rendering only).

[Deferred translucent lighting needs Lux Plus.](#)

Anisotropic lighting uses an anisotropic GGX BRDF as proposed by Brent Burley and allows you to define a specular spread along the tangent or bitangent in order to create materials like brushed metal or even hair.

Please note: You currently can not switch between different lighting models within a single shader so wetness and snow will produce pretty strange specular highlights when added on top of a material which uses anisotropic lighting.

Deferred anisotropic lighting needs Lux Plus.

Pre-integrated skin lighting is based on the work of Eric Penner. It adopts an implementation first presented by Farfarer and combines deep subsurface scattering from a static depth map (like translucent lighting) and light scattering calculated dynamically based on the given curvature of the (skinned) surface.

Deferred skin lighting needs Lux Plus.

Diffuse scattering, asperity scattering or peach fuzz currently simply boosts the albedo color at grazing angles but gives you a very cheap possibility to create surfaces like velvet or surfaces covered by a thin layer of “something” that scatters light like e.g. ice. Compared to translucent lighting this will only add light scattering at the outmost parts of the surface whereas translucent light will give you deep subsurface scattering.

Ambient specular reflections use Lazarov’s environmental BRDF optimized by Epic. You can return to the built in BRDF editing the “Lux Config.cginc”.

(Lux 2.02.: deprecated) Smoothly fading point light shadows in deferred only, sorry.

Specular Anti Aliasing caused by [geometry](#) and/or bump maps.

Further notes about anisotropic lighting

Lux anisotropic lighting model uses the tangential direction of the material’s grooves or strands to calculate how the light will be reflected by the surface.

Please note: When using anisotropic lighting Smoothness must never be 1.0 but should be clamped at 0.975. The provided example shaders do this by limiting the range input for Smoothness in the shader properties. When using smoothness texture however you have to author the texture accordingly.

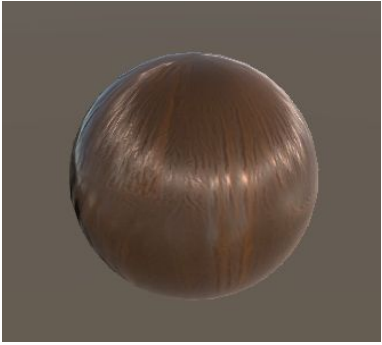
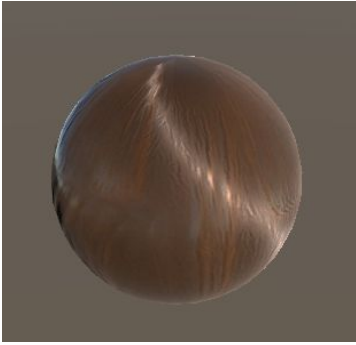
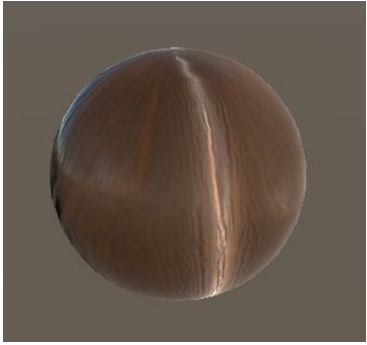
By default specular highlights are spread along the original tangent direction. But you may adjust this using:

Base Tangent Direction: A Value of 0.0 / 1.0 / 0.0 matches the original tangent direction whereas a value of 1.0 / 0.0 / 0.0 would describe the bitangent direction.

Strength: A value of 1.0 would make the shader use the original tangent direction (tweaked by Tangent Direction texture input, see below) whereas a value of 0.0 would make the shader use the specified Base Tangent Direction.

The example below shows how specular highlights will change if one shifts the final tangent direction passed to the lighting function from the original tangent direction (slightly shifted by

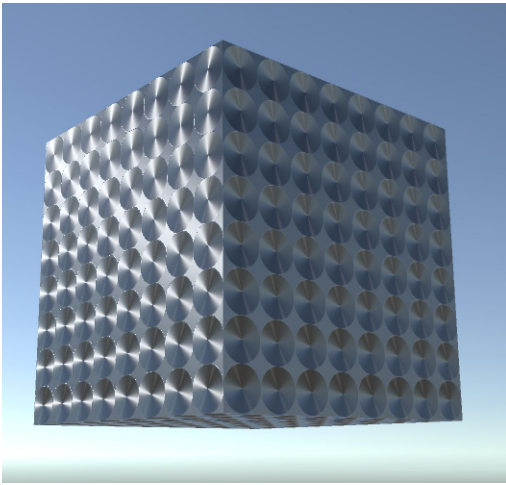
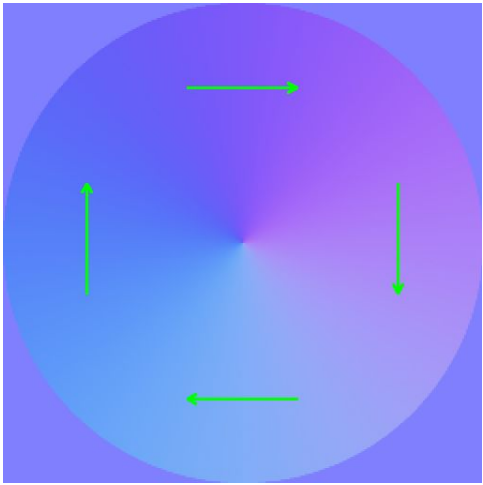
texture input: left) to the bitangent direction (no influence from the tangent direction texture any more: right):

		
Base Tangent Dir: 1.0 / 0.0 / 0.0 Strength: 1.0	Base Tangent Dir: 1.0 / 0.0 / 0.0 Strength: 0.85	Base Tangent Dir: 1.0 / 0.0 / 0.0 Strength: 0.0

In order to create effects like brushes metal you will have to assign a **Tangent Direction texture** which will tweak the tangent direction passed to the lighting function on a per pixel basis according to your uv layout.

That being said you should try to always align your uvs either on the u or the v axis to make things a bit easier to handle :-)

Tangent Direction textures should be authored and imported as “normal” texture. Please have a look at the provided demo content to find out more.

	
Brushed metal	Used Tangent Direction Texture: Green arrows show how the tangent direction is tweaked – always according to your uv layout and the original tangent direction though.

Anisotropic lighting always uses the **metallic setup** as it is also used for simple anisotropic hair shading – and using the metallic setup let us have at least “some” colored specular highlights on hair when it comes to deferred.

Please note: Deferred shading does not handle Metallic = 0.0 correctly (as it would cost some extra instructions). So you have to take care about this when authoring your metallic texture or setting up the metallic value using a slider.

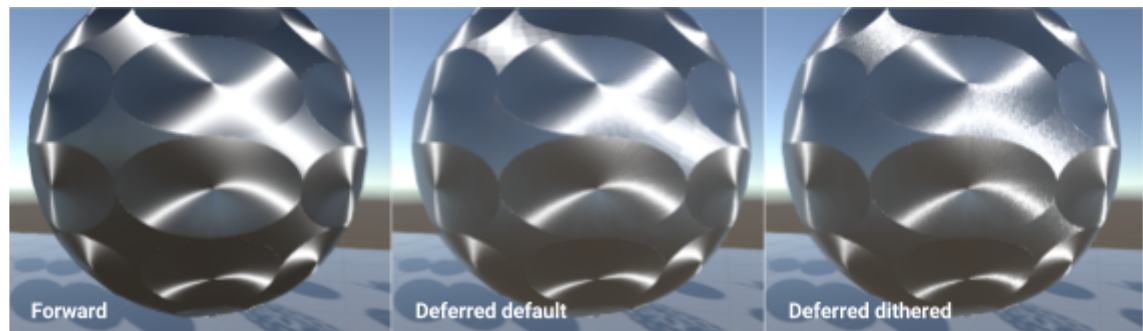
Also because of hair shading anisotropic lighting supports **translucent lighting** by default. You may simply turn it on or off. A simple translucency mask will be derived from the metallic mask. Distortion, Power, Scale and Shadow Strength for translucent lighting is controlled globally by the “Lux Setup” script.

Adding **diffuse scattering** when using the metallic workflow is not really correct as it will change the finally calculated specular color as well – but it looks ok on the provided hair shader :-).

Deferred anisotropic lighting

Deferred anisotropic lighting (**Plus only**) works more or less like forward rendered anisotropic lighting but comes with 2 main drawbacks due to the limited size of the built in gbuffer:

1. **Direct specular highlights** currently might look **quantized and suffer from banding artifacts**. This usually is not (very) visible – but in case you have very smoothly curved surfaces combined with just a constant smoothness, no bumps at all and a pretty close camera distance it might get noticeable.
You may address this issue by tweaking the material like adding bumps, using a lively smoothness texture or adding some noise to the tangent direction texture. Or you may activate **dithering** in the shader so that the tangent direction will slightly be changed from pixel to pixel.
In case none of the proposals fit your needs you may always use a forward only shader of course :-(. An example shader is included (Anisotropic Base Forward Only).



2. **Ambient specular reflections** currently are not “correct” and do not fit the reflections rendered in forward – nevertheless they give you an interesting and believable look. Like direct specular highlights they are prone to quantization.

Please pay attentions to these facts when authoring your art.

Please note: Anisotropic lighting has been reworked in 2.02 and now gives much smoother results in deferred, so dithering might not be needed anymore.

Specular Anti Aliasing (Lux 2.02)

The latest version of Lux lets you use specular anti aliasing when using deferred rendering in order to reduce rendering artifacts caused by tiny little highlights. As it does cost some instructions (although it is rather cheap) you may disable it by editing the "**Lux Config.cginc**" file.

Specular anti aliasing currently is only supported on standard lighting using deferred rendering.

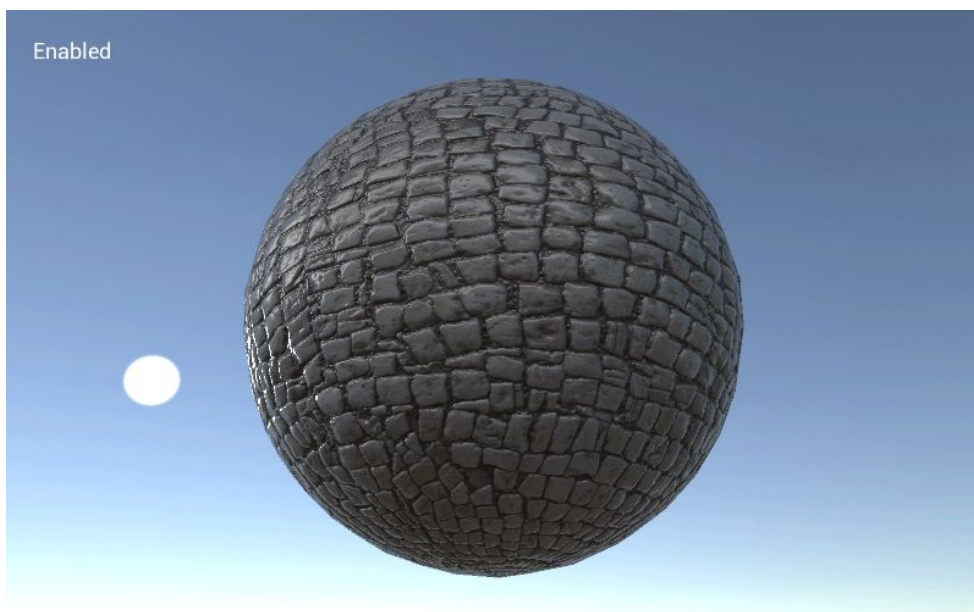


Please note: Specular Anti Aliasing only handles aliasing caused by detailed **geometry**. If you get artifacts caused by highly detailed bumps you should prefilter your smoothness textures using the [Lux TexturePostprocessor](#).

Horizon Occlusion (Lux 2.02)

Image based reflections might produce some kind of "light leaking" on normal mapped surfaces as the final reflection vector might point *behind* the actual surface resulting in reflections which simply are "impossible". Horizon Occlusion fixes this.

Read more here: <http://marmosetco.tumblr.com/post/81245981087>



Horizon Occlusion is enabled by default. You can disable it globally by editing the "**Lux Config.cginc**" file.

Please note: Custom surface shaders need *Specular Anti Aliasing* to be enabled as well to work properly.

Surface Features

Dynamic Weather

Dynamic weather consists of wetness/water and snow – both generally controlled by global script input such as temperature and rainfall over time as well as material and object specific properties like world normal, height or slope damp.

Snow

The snow distribution is calculated based on the amount of accumulated snow (driven by script input), the given y-position in world space, the surface normal in world space and the world normal damp factor. In order to create a more lively snow accumulation all this finally is combined with a globally defined snow mask which gets sampled twice using different frequencies: a low frequency to control the large scale snow accumulation and a high frequency in order to add details to the snow distribution mask.

You may even add a unique snow mask per material to mask e.g. footprints, skid marks or occluded areas under overhanging roofs.

- Snow lerps the albedo color towards the globally defined snow color.
- Snow smoothes the underlying surface normals and fades in the snow and snow detail normal. (If no underlying surface normal map is available snow accumulation might look a bit odd.)
- Snow lerps smoothness and specular towards smoothness (defined by global texture input) and the specular color (defined by script input) of snow.
- Snow extinguishes translucent lighting and emission according to its opacity.
- Melting snow will add wetness to the material. However freezing wetness right now simply fades out.

Wetness and Water

Water distribution is calculated based on the the amount of accumulated wetness and water (both driven by script input). Water accumulation takes the height of the given fragment into account which is provided by a height map and defines cracks in which water will accumulate first. Puddles are defined by a puddles mask.

Please note: If no height map is available water accumulation will look a bit strange. So materials supporting dynamic wetness and water will most likely also support parallax mapping.

Wetness and water supports dynamically blended rain ripples fitting the rain intensity as well as water flowing down the surface.

- Water will lerp smoothness and specular toward smoothness and specular of water (hard coded in the shader).
- It will lerp the normal towards a flat normal.

- If rain ripples and/or water flow are enabled water will add rain ripples and/or water flow which perturb the normal and refract the uvs which are used to finally sample all further textures like albedo, normal or smoothness.
- It will calculate the porosity of your material (based on smoothness and specular color) and darken the albedo according to the amount of accumulated water.
- It will lerp the albedo towards the given water color – which lets you add some kind muddyness to your materials: If “solid” materials like rock will most likely have a watercolor of 0,0,0,0 “smooth” materials like sand or mud might have a distinguished watercolor which will tint the albedo and suppress the underlying normals, translucent lighting and emission.
- It will lerp occlusion towards 1.0 according to the water color alpha value.

Mix Mapping

Unlike unity’s built in detail mapping mix mapping allows you to blend between 2 different texture sets either controlled using vertex colors or texture input and takes the height per pixel into account if present to calculate the final blending result.

It supports 2 independent physically based material sets of which each has its own albedo, diffuse scattering, smoothness, specular and normal as well as special wetness/water and snow settings.

Transparency and emission are always controlled by the primary material though.

Translucent lighting also is only supported on the primary material – the second material will be rendered as fully opaque.

Parallax Mapping and Parallax Occlusion Mapping

Both techniques let you to add a much better sense of depth to your materials. Whereas parallax mapping is super-cheap to render as it only needs one single additional texture lookup and some very easy math POM really might kill your frame rate as Lux uses a pretty simple but quite accurate implementation if it comes to mix mapping.

Please note: Using POM combined with alpha testing in forward rendering gets super-expensive as the shader has to do the POM calculations 3 times for the regular pass, the depth pass and the shadow caster pass.

All features described above (except from anisotropic lighting) are available using the provided Lux Standard Shader as well as writing your own custom surface shaders.

Using Lux

Setting up your Project

Before you can use Lux you will have set up your project properly.

In case you use deferred rendering you have to assign the “Lux Internal-DeferredShading” shader: Go to “Edit” -> “Project Settings” -> “Graphics” -> “Built in shader settings” -> “Deferred” and set it to “Custom”.

Then assign the “Lux Internal-DeferredShading” shader to the new slot (the shader is located in “Lux Shaders/Lux Core/Resources”).

Lux Plus only: You have to assign the “Lux Plus Internal-DeferredShading” shader and the custom deferred reflection shader “Lux Plus Internal-DeferredReflections” – both located in the folder: “Lux Shaders/Lux Core/Resources”.

Lux ships with a custom “Lux Plus DepthNormal” shader as well, which should be assigned too following the official 5.4.x documentation. However doing so will break image effects such as SSAO in Unity 5.4.2. So i recommend to simply restart Unity after import to make the “Lux Plus DepthNormal” shader overwrite the built in one.

Unity 5.5.0b10 does not offer the possibility to assign a custom “DepthNormal” shader anyway.

You should also make sure that your project uses linear color space (“Edit” -> “Project Settings” -> “Player” -> “Other Settings” -> “Color space”) and your camera is set to “HDR”.

Forward rendering and fog

Due to the vast amount of shader variants the Lux standard shaders and some nasty “bugs” in the Unity Editor and how it handles unused shader variants i had to skip some dynamic options within the Lux standard shaders. One of these is the automatic handling of the various fog types when it comes to forward rendering:

Lux standard shaders by default support **Exponential Squared** fog.

In case you would like to use linear or simple exponential fog and forward rendering you will have to edit the “**Lux Config.cginc**” file and comment/uncomment the corresponding defines.

Enabling exponential squared fog looks like this:

```
// #define FOG_LINEAR
// #define FOG_EXP
#define FOG_EXP2
```

Enabling linear fog instead would look like this:

```
#define FOG_LINEAR
// #define FOG_EXP
// #define FOG_EXP2
```

When done simply save the “Lux Config.cginc” file and reimport the Lux standard shaders.

Please note: You can not use different fog modes within a single project as far as the Lux standard shaders are concerned. Custom surface shaders however still handle all fog modes automatically.

Please note: Even if your camera is set to use the deferred rendering path some shaders might use forward rendering due to alpha blending. So you should define the desired fog mode even if you use deferred rendering.

Lux Plus and Image Effects

As Lux Plus packs Unity’s built in GBuffer in a special way, image effects, which rely on data from the GBuffer, will most likely break. The most common affected effects are:

- SSAO or HBAO

- Screen Spaces Reflections

For this reason Lux Plus ships with customized versions of Unity's Cinematic Image Effects and offers hacks for other popular effects.

Cinematic Image Effects

Effects of the Cinematic Image Effects which are affected are **Ambient Occlusion** and **Screen Space Reflections**. In order to make them work properly just remove these effects from the original folder (which should be "Standard Assets" -> "Effects").

Then import the package "Lux xEffects Cinematic Image Effects" from the "Lux Plus" folder. Now the tweaked image effects should be available under "Component" -> "Image Effects" -> "Cinematic" as "Ambient Occlusion" and "Screen Space Reflections".

Please note: The Cinematic Image Effects have to be present in your project to make the tweaked shaders work.

Please note: The Cinematic Image Effects are not supported by Lux Plus 2.02. Please use the Post Processing Stack instead.

Post Processing Stack

Just like the Cinematic Image Effects Unity's new Post Processing Stack shaders needs some minor tweaks to work properly with Lux Plus. In order to apply these tweaks follow the steps below:

- Move the folder "PostProcessing" into the Lux's root folder as the tweaked shaders have to include some Lux functions and otherwise won't find these.
- Delete the original "AmbientOcclusion" and "ScreenSpaceReflection" shaders.
- Import the "Lux xEffects Post Processing Stack" package. Then move its content to the "Shaders" folder within "PostProcessing" -> "Resources" – if the package not already extracted itself to that location.
- Done.

Please note: You have of course to import the PostProcessing suit first:

<https://github.com/Unity-Technologies/PostProcessing>

Amplify Occlusion

Amplify Occlusion offers different methods to sample the "Per Pixel Normals":

- In case you use "Camera" everything just should be fine.
- In case you use "G Buffer" you would have to edit Amplify's occlusion shader:

Open the "Occlusion" shader.

Find the following line (line 113 in version 1.1.0):

```
if ( source == NORMALS_GBUFFER_OCTA_ENCODED && gbuffer2.a < 1 )
```

And change it to:

```
if ( source == NORMALS_GBUFFER_OCTA_ENCODED || gbuffer2.a == 0 )
```

That is all! Do not use "G Buffer Octa Encoded".

Other Image Effects

Other image effects like "SSAO Pro" or "Horizon Based Ambient Occlusion" currently are not supported – but might follow in the future.

You can always use the Lux hacked Ambient Occlusion version of the Cinematic Image Effects instead.

Setting up your Scenes

Like i have mentioned above Lux has been written having deferred shading in mind. And as the size of unity's gbuffer is rather large but still limited not all shading parameters can be setup per material and some parameters have to be defined globally.

For this reason each scene which you would like to use Lux driven shaders in need the **"Lux Setup"** script to be present. [Read more >](#)

In case you want to use dynamic weather you have to also add the **"Lux Dynamic Weather"** script. [Read more >](#)

Please have a look at the provided demo scenes and refer to the dedicated chapters to find out more about these scripts.

When done you should be ready to go.

Using the Lux standard shader

When you have set up your project and scene you are ready to set up your first material using Lux. Start by creating a new material and select the "Lux Standard (Specular setup)" shader.

The Lux standard shader material inspector is based on the built in one so you should already know a lot of its parameters.

Rendering Mode

Allows you to define whether the material supports transparency, and if so, which type of blending mode to use.

Culling

Default is "Back", so all faces facing away from the camera will be culled. But you may choose "Front" for whatever reason. If culling is set to "Off" you may create a correctly lit double sided material from single sided geometry like foliage, windows or curtains by checking **Double Sided**.

Lighting

Lets you switch between standard and translucent lighting. Anisotropic lighting currently is not supported and might never be as it is very special.

Only available in Lux Plus.

Standard Lighting

Just the built in standard lighting.

Translucent Lighting

If you chose translucent lighting you will get two new parameters listed in the **Main Maps** section below “Occlusion”:

Translucency: Factor to adjust translucency (0.0 = opaque, 1.0 = fully translucent).

Scattering Power: Power value or exponent determining the view dependent falloff.

Please note: Other translucent lighting settings are controlled globally by script. If Mix Mapping is enabled Secondary Maps will be rendered as fully opaque.

If the **Scattering Power** is set to 0.0 the shader uses wrapped NdotL based translucent lighting – suitable for single sided geometry and foliage rendering.

In case you want to use a **texture to control translucency** strength you have to store it in the blue color channel of your **occlusion texture** and check **Combined Map**. We do not use the high quality alpha channel here in order to keep the texture smaller as far as memory consumption is concerned.

The texture value will be multiplied with the value from the Translucency slider.

Anisotropic Lighting

Currently not supported by the standard shader as it is still in development and very special. You can however use one of the “Lux custom Anisotropic Shaders” to add anisotropic lighting to your materials.

Mix Mapping

Enable Mix Mapping: If checked Secondary Maps will not be regularly added on top of the Main Maps like in the built in standard shader. Instead the shader blends between Main and Secondary Maps according to a mix map. By default the mix map will be taken from vertex color red.

Please note: If mix mapping is enabled Secondary Maps will allow you define further parameters like ambient occlusion (which should be stored in the alpha channel of the diffuse texture), diffuse scattering and specular.

Mix Mapping also influence most other features like dynamic water and snow – most likely adding further parameters too.

Use Detail Mask: If checked mix mapping will be controlled by:

- a) the **green** color channel of the **Detail Mask**.
- b) the **blue** color channel of the **Height Map** in case you have assigned a height map.
If a Height Map is assigned the shader expects 2 different height maps for Main and Secondary Maps in the color channels green and alpha and supports much more lively height based blending.

The sections **Main Maps** and **Secondary Maps** more or less fit the ones from the built in standard shaders and let you declare the main material properties and details which get overlayed on top of the base textures.

But if you have a closer look at the Main Maps section you will notice a new feature which is Diffuse Scattering.

Diffuse Scattering

Diffuse Scattering: Lets you define a diffuse scattering color. If set to black the shader skips diffuse scattering.

Scatter Bias: Lets you raise the diffuse light scattering by adding a constant.

Scatter Power: Lets you sharpen the diffuse light scattering towards grazing angles.

Parallax Occlusion Mapping

As soon as you add a height map to the Main Maps the inspector will show additional parameters:

Lux 2.02 UV Ratio (XY) Scale (Z): As parallax occlusion mapping is calculated in tangent or texture space the final effect is directly connected to the width to height ratio of your textures and the uv layout.

UV Ratio (XY) When using squared textures and more or less and evenly uvs the *UV Ratio* is simply 1, 1.

When using a texture with e.g. a width of 256px and a height of 2048 and evenly uvs the *UV Ratio* would be 1, 8. In case you did not adjust the *UV Ratio* the extrusion would somehow look distorted or stretched along the V-axis.

So use *UV Ratio* in order to relax the final result.

Scale (Z): In case you enable *rain ripples* or use *world space mapped snow* (see below) the shader has to transfer the final extrusion from tangent space to world space so it needs to know the ratio between uvs and world space.

Taking a built **primitive cube** scale would be 1 as the standard cube is 1x1x1m in size while its uvs also cover the 0-1 range.

Using the built in **primitive plane** however would need you to adjust the scale to 10 as the plane is 10x10m in size while its uvs are in the 0-1 range.

Please note: Adjust the Scale parameter by activating world space mapped snow. Then change its value until the snow exactly matches the extruded surface and does not float when you orbit the camera.

Lux 2.01 Parallax Tiling: Specify the tiling of the height map relative to the base UVs. Default is 1. This mostly gets interesting in case you use mix mapping and store the mix mapping mask in the blue color channel: In order to break up tiling artifacts you might consider tiling the heightmaps already in the texture (e.g. 2x2 – you may even add some variation here) but use a 1x1 mask to reduce visible tiling patterns.

Lux 2.02 Mask Tiling: *Parallax Tiling* has been renamed to Mask Tiling: The tiling value actually does not affect the height map samples anymore. Only the mask gets sampled using a different tiling value.

Enable POM: Check this to use POM

Linear Steps: Max number of linear steps performed while raycasting the heightmap. Higher numbers will give you more accurate results but are more expensive.

Please note: Especially if you use POM it is essential that you setup the import settings of the heightmap properly: It should be set to "Advanced" and "Bypass sRGB sampling" *checked* or *unchecked* "sRGB (Color Texture)" in case you use Unity > 5.4. I guess – as otherwise you may get strange discontinuities on the extruded surface.

Dynamic Weather

Snow: If enabled the shader will add dynamically accumulated snow.

A lot of the dynamic snow parameters are controlled by script input (please have a look at the section about the "[Lux Dynamic Weather](#)" script) – such as current snow amount, the used snow mask, snow normal and snow detail texture.

But each material lets you set up different snow parameters as well:

Lux 2.02 Snow Mapping: You may choose between *Local Space* and *World Space*. While *Local Space* will sample the snow masks and detail textures using the model's uvs, *World Space* will sample the snow textures using a top down projection in world space as you may know from the terrain shaders. This options lets you easily cover large areas containing various objects like roads, sidewalks and a terrain with a "uniform" and continuous snow distribution.

Snow Slope Damp: Determines the influence of the world normal up vector on snow accumulation: A value of 0.0 would accumulate snow even on faces pointing downwards whereas higher values will shrink the snow accumulation towards faces pointing upwards.

Snow Accumulation

Material Constant: Lets you add a constant amount of snow to the material – regardless of temperature or rainfall. "Snow start height" however will be taken into account.

Global Influence: Factor for the globally controlled snow accumulation which lets you e.g. speed up or slow down / clamp snow accumulation for the given material.

Snow Tiling: Tiling factor for the global snow texture relative to the base texture.

Snow Normal Strength: Strength of the global snow normal texture.

Snow Mask Tiling: Tiling factor for the global snow mask texture relative to the base texture.

Snow Detail Tiling: Tiling factor for the global snow detail texture relative to the base texture.

Snow Detail Strength: Strength of the global snow detail normal texture.

Snow Opacity: Lets you suppress emission and translucent lighting under snow.

Wetness: If enabled the shader will add dynamically accumulated wetness and water.

As for snow water accumulation and wetness rendering is driven by script input as well as by material specific parameters. The settings per material are:

Water Slope Damp: Determines the influence of the world normal up vector on water accumulation: A value of 0.0 would accumulate water and wetness even on faces pointing downwards whereas higher values will shrink the water accumulation towards faces pointing upwards.

Water Color: Defining a specific water color lets you add effects like muddiness. RGB controls the final color whereas Alpha controls the opacity. Colored water will suppress emission and translucency based on the given alpha value.

Water Accumulation in Cracks

Cracks are defined by a height or height map. Lower parts will be flooded first while higher parts will be flooded later.

Material Constant: Lets you add a constant amount of wetness to the material – regardless of temperature or rainfall.

Global Influence: Factor for the globally controlled water accumulation which lets you e.g. speed up or slow down / clamp water accumulation for the given material.

Water Accumulation in Puddles

Water in Puddles usually accumulates and vanishes slower (defined by script input). Puddles might be defined using vertex colors as well as texture input.

Material Constant: See above.

Global Influence: See above.

In case water flow is enabled:

Flow Normal Tiling: Tiling factor for the global water flow texture relative to the base texture.

Flow Speed: Speed of water running down the surface.

Flow Interval: The shader lerps between 2 offsetted and looped water flow animations. So this parameter determines the duration of each of the 2 loops.

Flow Refraction: Amount of refraction the flow normals add to the final texture look ups.

Flow Normal Strength: Effects the final normal getting written out for the lighting function but also influences “Flow Refraction”

Please note: Unlike “*Water Accumulation in Cracks*” and “*Water Accumulation in Puddles*” the thin layer of “*Wetness*” can’t be controlled per material but always uses the value which gets passed by the “*Dynamic weather*” script.

Lux Setup Script

The Lux Setup script provides the shaders with all parameters that can not be setup per material but have to be declared globally.

Detail Distance: Distance in which “small scale” surface details like parallax occlusion mapping, water ripples and water flow or detail snow bumps are rendered.

Detail Fade Range: Range over which details will fade out.

Enable Area Lights: Lets you globally enable and disable area lights.

Please note: As all BRDFs support translucent lighting you will find a couple of corresponding parameters for each BRDF – just to give you a bit more freedom : -)

Translucent Lighting

Power: *Power value or exponent for direct translucency or large scale subsurface scattering determining the view dependent falloff – is defined per material.*

Distortion: Defines subsurface distortion by shifting the light direction according to the surface normals, view dependent.

Scale: Scales deep subsurface light transportation or translucency.

Shadow Strength: Lets you suppress shadows on translucent parts.

Shadow Strength NdotL: Lets you define a specific shadow strength for materials using NdotL based translucency like foliage.

Skin Lighting

BRDF Texture: Lookup texture for diffuse skin lighting. Please use the provided one. If this texture is missing skin lighting will look completely broken.

Subsurface Color: Color of large scale subsurface light scattering.

Power: Power value or exponent for direct translucency or large scale subsurface scattering determining the view dependent falloff:

Distortion: Defines subsurface distortion by shifting the light direction according to the surface normals, view dependent.

Scale: Scales deep subsurface light transportation or translucency.

Enable Skin Lighting Fade: When checked skin lighting lerps towards standard lighting based on the distance to the camera. This lets you swap out the shader at lower LODs so you might use the simple "Lux/Human/Skin Standard Lighting" shader or even create a unified material for your characters using the (Lux) standard shader.

Skin Lighting Distance: Distance at which the shader shall not do any skin related lighting.

Skin Lighting Fade Range: Range starting from "Skin Lighting Distance" in which the shader performs the blending.

Anisotropic translucent Lighting

As anisotropic lighting also supports translucency you may specify:

Power: Power value or exponent for direct translucency or large scale subsurface scattering determining the view dependent falloff.

Distortion: Defines subsurface distortion by shifting the light direction according to the surface normals, view dependent.

Scale: Scales deep subsurface light transportation or translucency.

Lux Dynamic Weather Script

This script currently is just a rough draft although it does its job already quite well.

It basically provides the shaders with all needed inputs to render dynamically accumulated water and snow like temperature, current rainfall or the amount of already accumulated water as well as the needed textures.

Script controlled Weather: If checked water and snow accumulation will be calculated and set automatically based on the given temperature and amount of rainfall.

In case you are working on a "static" environment uncheck this to be able to tweak wetness, snow accumulation and "Water to Snow" manually.

Time Scale: Lets you globally control the speed of all effects like water accumulation in cracks so you do not have to tweak each single value..

Temperature: Values below 0.0 will fade rain and wetness into snow. Values above 0.0 will melt snow and dry water in case it is not raining.

Rainfall: The amount of currently falling rain or snow.

Wetness describes a “thin layer of wetness” which gets added to the whole surface regardless of any height (unlike water in cracks or puddles). Usually it should accumulate faster than water in cracks or puddles but dry slower.

Wetness Influence On Albedo: Lets you adjust the darkening of the albedo according to the thin layered wetness.

Wetness Influence On Smoothness: Lets you adjust the influence of the thin layered wetness on smoothness.

Accumulation Rate Wetness: Lets you adjust, how fast wetness will accumulate.

Evaporation Rate Wetness: Lets you adjust, how fast wetness will dry.

Accumulation Rate Cracks: Lets you adjust, how fast water will accumulate in small cracks – usually defined by a height map.

Accumulation Rate Puddles: Lets you adjust, how fast water will accumulate in puddles – usually defined by a height map and a puddle mask.

Evaporation Rate Cracks: Lets you adjust, how fast water will dry in small cracks.

Evaporation Rate Puddles: Lets you adjust, how fast water will dry in puddles.

Accumulation Rate Snow: Lets you adjust, how fast snow will accumulate.

Accumulated Wetness: Amount of wetness accumulated. Script controlled or manually set.

Accumulated Cracks: Amount of water accumulated in Cracks. Script controlled or manually set.

Accumulated Puddles: Amount of water accumulated in Puddles. Script controlled or manually set.

Accumulated Snow: Amount of snow accumulated. Script controlled or manually set.

Water to Snow: Actually drives the blending from water/wetness to snow and vice versa. melting of snow. Script controlled or manually set.

Please note: If Water to Snow is set to 0.0 (water only) you will not get any snow even if the amount of accumulated snow is set to 1.0. If it is set to 1.0 (snow only) you won't see any rain ripples nor water flow.

Water to Snow Time Scale: Lets you define the time it takes to melt snow when temperatures are $> 0.0^{\circ}$. Higher temperatures will make snow melt faster.

Water To Snow Curve: Curve that drives melting snow/ freezing water.

Snow Color: Global albedo color of snow.

Snow Specular Color: Global specular color of snow.

Snow Scatter Color: Global diffuse scattering color of snow – if diffuse scattering is supported by the shader.

Snow Diffuse Scattering Bias: Lets you raise the diffuse light scattering by adding a constant.

Snow Diffuse Scattering Contraction: Lets you sharpen the diffuse light scattering towards grazing angles.

Snow Mask: Texture which controls snow accumulation and stores the detail snow normal.

- Snow mask is stored in color channel **Blue**.

- It also contains the detail snow normal stored in **Green** and **Alpha**: So if you have a regular normal texture copy its Red channel to the Alpha channel while the Green channel stays untouched.

Snow Water Bump: Shared normal map for large scale snow (according to “Snow Tiling” as set up in your material) and water flow which is running down the surfaces which also contains the smoothness texture for snow.

- Normal is stored in **Green** and **Alpha** (Red from the original normal map goes into Alpha, Green into Green).

- Snow Smoothness is stored in **Blue**.

Please note: Both textures mentioned above should be imported using “Texture Type: Advanced” -> “Import Type: Default” -> “Bypass sRGB Sampling: checked”.

You may even think of importing the Snow Water Bump texture as uncompressed texture to improve the quality of the final normal and get rid of quantization artifacts. Lower the texture size in order to get its foot print rather small.

The example texture provided with the package are more or less only placeholders. You most likely can do much better...

Snow Start Height: Defines the height in world space above which snow is added. Please note that even materials which have a constant snow value assigned to will respect this.

Snow Height Blending: Lets you smooth out the border between faces covered by snow and not covered by snow.

Snow Particle System: Assign your snow particle system here in case you want it to be controlled by the script.

Max Snow Particle Emission Rate: Lets you define how many particles should be emitted if rainfall is set to 1.0 and temperature is below 0.0.

Rain Ripples: Rain ripple texture. Please assign the provided one (Animated Water Ripples) unless you know what you do.

Ripple Tiling: Size of the rain ripple texture in world space units (meters).

Ripple Anim Speed: Lets you control the speed of the rain ripple animation.

Ripple Refraction: Lets you define how much uvs get refracted by the ripple normals.

Please note: Rain ripples are always sampled in world space. So their scale will be constant over all objects no matter if latter are scaled or not. Rain ripples use a simple top down projection so they get distorted at steep angles.

Rain Particle System: Assign your rain particle system here in case you want it to be controlled by the script.

Max Rain Particle Emission Rate: Lets you define how many particles should be emitted if rainfall is set to 1.0 and temperature is above 0.0.

Snow GI master renderers: As accumulated snow will dramatically change the way light bounces you may assign some master renderers here which will be watched and trigger enlighten to recalculate GI if needed.

Size: You will need one master renderer per GI System. So set size to the number of your GI systems.

Elements: Then assign one master renderer per GI System.

Using diffuse fill lights

Diffuse fill lights allow you to add point or spot lights which mainly affect diffuse lighting while specular highlights might be completely suppressed or simply dimmed in order to create "traditional" fill lights and fake global illumination.

Making a regular point or spot light to be rendered as diffuse fill light is super simple as all you have to do is tweaking the light color's alpha value – no script needed.

In case you use forward rendering diffuse fill lights work on Lux shaders only. In case you use deferred rendering it works on all shaders.

Using Area Lights

Lux area lights are based on the work by Brian Caris for the Unreal Engine and are just a cheap approximation. That being said area lights are prone to cause some shading artifacts – especially on super smooth surfaces not using any bump maps like shown in the area lights demo.

Simply adding some gently bumps however will most likely solve these problems.

Area Lights are supported on all shaders when using deferred: No matter if you use the built in standard shader or any custom third party surface shader: As soon as you assign the "Lux Internal-DeferredShading" shader, this shader will take care.

In case you use forward rendering or shaders which are forward only shaders will NOT support area lights unless you move them to the appropriate "Lux custom (Lighting) Shaders" folder and make them use the lux lighting functions (the Lux Standard shader of course as well as the provides sample surface shaders support area lights even in forward).

In order to enable area lights you will have to check the proper checkbox in the "Lux Setup" script and add the "Lux Area Light" script to any light source you want to be rendered as area light (point or spot lights only).

Lux Area Light Script

Light Length: If set to values > 0.0 point lights will be rendered as tube lights.

Light Radius: If set to values > 0.0 point lights will be rendered as sphere lights and spot lights will be rendered as disc lights. Also set the light length in case you want point light to be rendered as tube lights.

Specular Intensity: Lets you extinguish specular highlights for the given light to create “diffuse fill lights”.

Important: When using the area light script you must not use the light component’s original light color and intensity properties to adjust (or animate) your light but the following parameters of the script:

Light Color: Color of the light overwriting the original one.

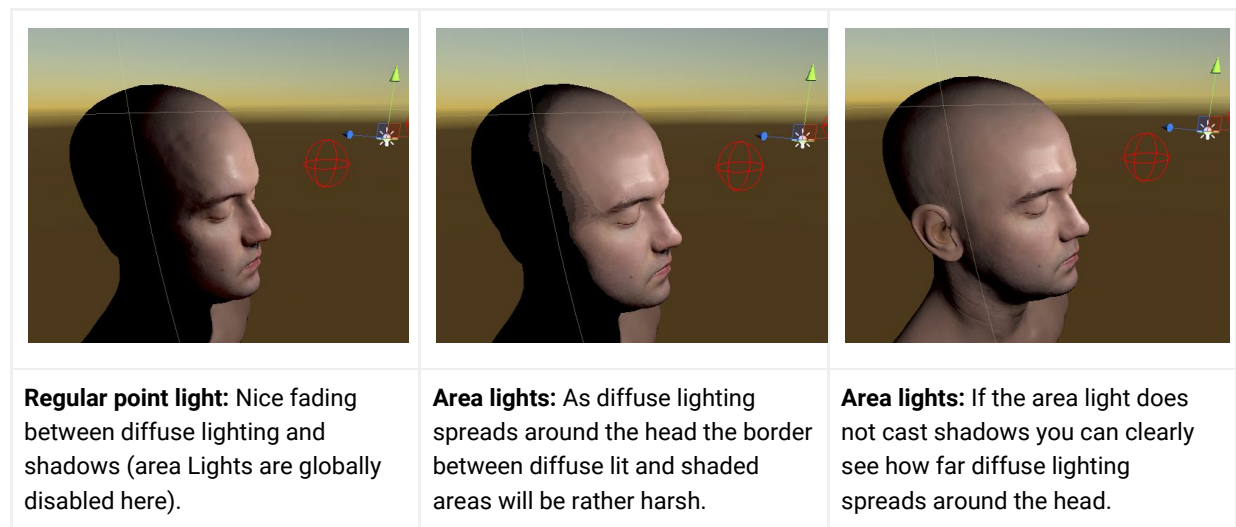
Light Intensity: Intensity of the light overwriting the original one.

As area lights are more or less useful only in indoor environments where you have artificial light sources such as neon tubes lights you can enable/disable the support for area lights globally even at runtime.

When rendering huge outdoor scenes with just the sun light active you should disable area lights to speed up rendering.

Area Lights and Shadows

Area Lights and self shadowing by real time shadows might look a bit strange as shadows are simply calculated for the original light source (point or spot light). So if you have a pretty large e.g. tube light the border between light and shadowed pixels might look pretty harsh.



Using the Lux TexturePostprocessor

The Lux Texturepostprocessor lets you prefilter your smoothness textures so aliasing on very smooth surfaces becomes (at least a little bit) less noticeable.

The script acts as AssetPostprocessor and will automatically look for textures that fit the naming conventions mentioned below and postprocess them whenever they are updated.

In order to be able to post process your smoothness texture – or better: the alpha channel of the "Specular Color (RGB) Smoothness (A)" or "Metallic (R) Smoothness (A)" texture the script needs the corresponding normal texture as well. For this reason:

- both texture have to be placed in the same folder.
- both textures have to have the same size.
- both textures have to be marked as "readable" (Import settings --> "Texture Type: Advanced" --> "Read/Write Enabled").
- both texture have to have the same file extension (e.g. ".tga" or ".psd").
- both texture have to be named properly:
 - Spec/Smoothness map: "YourtextureName_LuxSPEC"
 - Normal map: "YourtextureName_LuxNRM"

Writing custom Surface Shaders

Introduction

Why writing custom surface shaders if there is a pretty flexible Lux standard shader?

The answer is pretty simple: The Lux standard shader is not as flexible as you might want it to be: In order to not create loads of shader permutations, overload the interface and define more keywords than are currently used i had to limit its settings.

And as you will most likely much better know what your materials need or how textures would be packed efficiently i worked on giving anybody the possibility to write custom surface shaders which may fully benefit from Lux features.

I tried to make this as easy as possible and defined a bunch of macros which do most of the job. Nevertheless you will have to do some adjustments to your surface shaders depending on the features you want to use.

When writing custom surface shaders you first and foremost have to make sure that **your shader is located in one of the provided folders** according to the lighting function you want to use, as these folders contain a custom "AutoLight.cginc" which will tweak the outputs to the chosen lighting function's needs. The folders are: "Lux custom Anisotropic Shaders", "Lux custom Standard Shaders" and "Lux custom Translucent Shaders".

Next you will have to **edit the shader code** itself by adding the needed properties, keywords, includes, input definitions, vertex and surface shader functions...

Please note that Lux always relies on the **specular workflow**. Nevertheless you may use the **metallic workflow** within your custom surface shader block but you must ensure that all outputs are converted to fit the specular workflow before calling any lux surface modification function like "LUX_APPLY_DYNAMICWEATHER".

Do so by adding "LUX_METALLIC_TO_SPECULAR" at the end of your custom surface shader code.

But let's start with a simple shader using standard lighting (metallic workflow!) and add the support for area lights.

Adding support for Area Lights

As area lights do not need any special inputs or additional surface properties all we have to do is to make the shader use the Lux standard lighting model by including the corresponding lighting function. As Lux is based on the specular workflow but our example shader uses the metallic workflow we also have to include the “LuxUtils.cginc” and finally call “LUX_METALLIC_TO_SPECULAR”.

```
Shader "Custom/NewSurfaceShader" {
    Properties {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _Glossiness ("Smoothness", Range(0,1)) = 0.5
        _Metallic ("Metallic", Range(0,1)) = 0.0
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
        #pragma surface surf LuxStandardSpecular1 fullforwardshadows
        #pragma target 3.0
        #if defined (UNITY_PASS_FORWARDBASE) || defined(UNITY_PASS_FORWARDADD)
            #pragma multi_compile __ LUX_AREALIGHTS2
        #endif
        #include "../Lux Core/Lux Lighting/LuxStandardPBSLighting.cginc"3
        #include "../Lux Core/Lux Utils/LuxUtils.cginc"4
        sampler2D _MainTex;
        struct Input {
            float2 uv_MainTex;
        };
        half _Glossiness;
        half _Metallic;
        fixed4 _Color;

        void surf (Input IN, inout SurfaceOutputLuxStandardSpecular5 o) {
            // Albedo comes from a texture tinted by color
            fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            // Metallic and smoothness come from slider variables
            o.Metallic = _Metallic;
            o.Smoothness = _Glossiness;
            o.Alpha = c.a;
            // Convert metallic to specular output
            LUX_METALLIC_TO_SPECULAR6
        }
    }
    ENDCG
}
```

¹ **Declare the custom Lighting model** like you would declare any other lighting model.

² Enable Lux area lights using the related **keyword** and “multi_compile”, “shader_feature” or a simple “#define”.

³ **Include the custom lighting model.**

⁴ **Include the Lux Utilities** – needed because the shader uses the metallic workflow.

⁵ Make sure that the surface function uses our custom **surface output structure**.

⁶ Finally you have to **convert the outputs** to fit the specular workflow as this is the one Lux uses.

```

    }
    FallBack "Diffuse"
}

```

Adding Lux surface features

Adding Lux surface features however is more complicated as this will need you to:

1. define additional shader **Properties**.
2. include the needed **Lux functions**.
3. define additional **vertex to fragment Inputs**.
4. write a custom **Vertex function** to fill the Input structure.
5. tweak the **Surface function**.
6. enable the wanted features by setting the corresponding **keywords**.

So let's add "Dynamic weather" which is by far the most complex surface feature.

Shader Properties

Each Lux surface feature needs its own set of additional shader properties.

Dynamic weather e.g. needs the water color, flow speed, snow texture tiling and more.

You may simply copy & paste the needed properties from the "LuxShaderProperties.txt" file.

Please note: You do not have to specify the additional properties as material inputs (like `sampler2D snowMask`;) as this is done by the include files.

The Input Structure

The input structure contains the uv coordinates and other "standard" values provided by unity's shader compiler as well as Lux specific values.

In order to save texture interpolators we will have to pack our values like

`float4 lux_worldPosDistance;` Using `float2 lux_flowDirection;` would allow us to store another `float2` value by making it a `float4` before the number of available texture interpolators exceeded.

Another important aspect is the fact that we can not use "`uv_MainTex`" to declare the regular UVs as lux needs access to "`_MainTex_ST`". So we simply write to "`lux_uv_MainTex`".

In case you reach the limit of available texture interpolators you will have to shift some calculations from the vertex to the fragment shader :-(The struct we need for our example is:

Lux 2.01:

```

struct Input {
    float2 lux_uv_MainTex;           // We may not use uv_MainTex here!!!!
    float3 viewDir;
    float3 worldNormal;
    INTERNAL_DATA
    // Lux
    fixed4 color : COLOR0;           // make sure color is mapped to COLOR0 semantics
    float4 lux_worldPosDistance;     // needed by Water_Ripples and all detail effects
    // Lux dynamic weather
    float2 lux_flowDirection;        // needed by Water_Flow
};

```

Lux 2.02:

```

struct Input {
    float2 lux_uv_MainTex;           // We may not use uv_MainTex here!!!!
    float3 viewDir;
    float3 worldNormal;
    float3 worldPos;
    INTERNAL_DATA
    // Lux
    fixed4 color : COLOR0;           // make sure color is mapped to COLOR0 semantics
    float2 lux_DistanceScale;         // needed by various functions
    float2 lux_flowDirection;        // needed by Water_Flow
};

```

The Vertex Function

The custom vertex function fills all declared members of the input structure.

Make sure that you call it in the initial #pragma surface directive:

```
#pragma surface surf LuxStandardSpecular fullforwardshadows vertex:vert
```

Lux 2.01:

```

void vert (inout appdata_full v, out Input o) {
    UNITY_INITIALIZE_OUTPUT(Input,o);
    // Lux
    o.lux_uv_MainTex.xy = TRANSFORM_TEX(v.texcoord, _MainTex);
    o.color = v.color;
    // Calc Tangent Space Rotation
    float3 binormal = cross( v.normal, v.tangent.xyz ) * v.tangent.w;
    float3x3 rotation = float3x3( v.tangent.xyz, binormal, v.normal.xyz );
    // Store flow direction
    o.lux_flowDirection = ( mul(rotation, mul(_World2Object, float4(0,1,0,0)).xyz) ).xy;
    float3 worldPosition = mul(_Object2World, v.vertex);
    // Store world position and distance to camera
    o.lux_worldPosDistance.xyz = worldPosition;
    o.lux_worldPosDistance.w = distance(_WorldSpaceCameraPos, worldPosition);
}

```

Lux 2.02:

```

void vert (inout appdata_full v, out Input o) {
    UNITY_INITIALIZE_OUTPUT(Input,o);
    // Lux
    o.lux_uv_MainTex.xy = TRANSFORM_TEX(v.texcoord, _MainTex);
    o.color = v.color;
    // Calc Tangent Space Rotation
    float3 binormal = cross( v.normal, v.tangent.xyz ) * v.tangent.w;
    float3x3 rotation = float3x3( v.tangent.xyz, binormal, v.normal.xyz );
    // Store flow direction
    o.lux_flowDirection = ( mul(rotation, mul(_World2Object, float4(0,1,0,0)).xyz) ).xy;
    float3 worldPosition = mul(_Object2World, v.vertex);
    // Store distance to camera and object scale
    float3 worldPosition = mul(unity_ObjectToWorld, v.vertex);
    o.lux_DistanceScale.x = distance(_WorldSpaceCameraPos, worldPosition);
    o.lux_DistanceScale.y = length( mul(unity_ObjectToWorld, float4(1.0, 0.0, 0.0, 0.0)) );
}

```

The Surface Function

The most interesting part of course is the surface function which contains your regular surface definition and some Lux specific macros:

```
void surf (Input IN, inout SurfaceOutputLuxStandardSpecular o) {
    // Initialize the Lux fragment structure
```

Lux 2.01:

```
LUX_SETUP(IN.lux_uv_MainTex, float2(0,0), IN.viewDir, IN.lux_worldPos.xyz,
IN.lux_worldPos.w, IN.flowDirection, IN.color)
```

Lux 2.02:

```
LUX_SETUP(IN.lux_uv_MainTex, float2(0,0), IN.viewDir, IN.worldPos,
IN.lux_DistanceScale.x, IN.flowDirection, IN.color, IN.lux_DistanceScale.y)
// As we want to to accumulate snow according to the per pixel world normal we have
to get the per pixel normal in tangent space up front using the base uvs
// This will lead to the normal getting sampled twice :-( but that is how things
work.
o.Normal = UnpackNormal(tex2D(_BumpMap, IN.lux_uv_MainTex));
// Initialize dynamic weather which will calculate and store water and snow
accumulation in the Lux fragment structure which is used by LUX_APPLY_DYNAMICWEATHER
to finally tweak all outputs
LUX_INIT_DYNAMICWEATHER_TANGENTNORMAL(1, 1, o.Normal)

// Do your regular stuff
// Albedo comes from a texture tinted by color
fixed4 c = tex2D (_MainTex, lux.finalUV.xy) * _Color;
o.Albedo = c.rgb;
// Metallic and smoothness come from slider variables
o.Metallic = _Metallic;
o.Smoothness = _Glossiness;
// We have to sample the normal a second time using the final uvs as they might be
refracted by wetness
o.Normal = UnpackNormal(tex2D(_BumpMap, lux.finalUV.xy));
o.Alpha = c.a;

// Convert metallic to specular output
LUX_METALLIC_TO_SPECULAR
// Finally apply dynamic weather
LUX_APPLY_DYNAMICWEATHER
}
```

Initialize the Lux fragment structure

Lux surface features need the **Lux fragment structure** to be initialized. This structure defines and stores all needed values and can be easily accessed by all following functions as well as your own code.

Use the defined macro to initialize the Lux fragment structure:

Abstract:

```
LUX_SETUP(
    float2 main UVs,
    float2 secondary UVs,
    half3 view direction in tangent space,
    float3 world position,
```

```

        float   distance to camera,
        float2  flow direction,
        fixed4  vertex color
        //Lux 2.02
        float object scale
    )

```

Code that fits our custom input structure:

```

LUX_SETUP(
    IN.lux_uv_MainTex,
    float2(0,0),          // not needed in our example so it is set to 0,0
    IN.viewDir,
    // Lux 2.01
    IN.lux_worldPosDistance.xyz,
    IN.lux_worldPosDistance.w,
    // Lux 2.02
    IN.worldPos,
    IN.lux_DistanceScale.x,
    IN.lux_flowDirection,
    IN.color
    // Lux 2.02
    , IN.lux_DistanceScale.y
)

```

Initialize Dynamic Weather

As Dynamic Weather might refract the uvs used to sample e.g. the albedo texture we have to call it before we can do the regular texture look ups.

Calling `LUX_INIT_DYNAMICWEATHER` will calculate the water and snow accumulation and return the final uvs which might be refracted by water ripples or water flow.

Abstract:

```

LUX_INIT_DYNAMICWEATHER (
    half    puddle mask value,
    // may be defined by vertex color or texture input // [0-1 range]
    half    snow mask value,
    // may be defined by vertex color or texture input // [0-1 range]
    half3   tangent space normal
    // set this to half(0,0,1) in case you do not sample the normal texture up
    front
)

```

Code that fits our example:

```

LUX_INIT_DYNAMICWEATHER_TANGENTNORMAL(
    1,          // we do not use any specific puddle mask
    1,          // we do not use any unique snow mask
    o.Normal    // the sampled per pixel normal in tangent space
)

```

Please note: In case your shader shall support **single sided geometry** you will have to call `LUX_INIT_DYNAMICWEATHER_SINGELSIDED` which expects a fourth input parameter: `float3 flipFacing`.

You also have to declare a special keyword in order to make the shader compile properly:

```
#define EFFECT_HUE_VARIATION
```

Please have a look at the [Lux/Standard Lighting/Full Features DoubleSided Cutout](#) shader to find out more.

Do your regular stuff

When doing the regular stuff it is important to use the uvs returned by the dynamic weather function instead of the base uvs. It writes the final uvs to the Lux fragment structure so you can access them using: `lux.finalUV.xy`

Please note: `lux.finalUV` is a float4 value where xy store the refracted main uvs, zw the refracted secondary uvs.

Convert metallic to specular output

As the following function will tweak the albedo and specular values we will have to convert the outputs from the metallic workflow to the specular one by adding `LUX_METALLIC_TO_SPECULAR`.

Finally apply dynamic weather

Adding the macro `LUX_APPLY_DYNAMICWEATHER` will lerp all outputs like albedo, smoothness, specular, normal and emission towards wetness and/or snow according to the calculated water and snow accumulation.

The Keywords

In order to make all the changes finally get applied you will have to declare the corresponding keywords.

In order to use all features Dynamic Weather provides you would have to declare:

```
#define _SNOW
#define _WETNESS_FULL
```

before you include the Lux surface functions (please have a look at the provided surface shaders).

Skip `#define _SNOW` if you do not need any snow.

Or tweak the water/wetness rendering using the following keywords:

<code>_WETNESS_SIMPLE</code>	no ripples, no water flow but correct water accumulation
<code>_WETNESS_RIPPLES</code>	correct water accumulation and dynamic ripples
<code>_WETNESS_FLOW</code>	correct water accumulation and dynamic water flow
<code>_WETNESS_FULL</code>	correct water accumulation and dynamic ripples and water flow

So in case you create a shader for e.g. some rocks which will be populating a cave and you want to give them a wet look and even add water running down the surface `_WETNESS_FLOW` would be the right keyword for this as there will hardly be any rain inside the cave so water ripples are not needed at all.

Flat surface do not need any water running down so `_WETNESS_RIPPLES` might be the right keyword for these materials.

In case you do not want or need any wetness on your material do not declare any of the wetness keywords and the shader will only accumulate snow – and simple wetness based on melted snow.

Adding Mix Mapping

In case you want to use Lux mix mapping this always should be done right after you have set up the Lux fragment structure as it will drive all other functions like parallax extrusion or dynamic weather.

Currently there is no dedicated shader macro for mix mapping just due to the fact that it might be very special: You could do mix mapping according to vertex colors, texture input or any other parameter you can think of.

Mix mapping consists of 2 parts:

1. Setting up the mix mapping value `lux.mixmapValue` (half2)
2. Using the mix mapping value to blend between the two texture sets.

1. You can simply set up the mix mapping value by writing to the Lux fragment structure:
`lux.mixmapValue = half2(valueForTex0, 1.0 - valueForTex0);`
 So when using `vertex.color.red` e.g. it would be:
`lux.mixmapValue = half2(In.color.r, 1.0 - IN.color.r);`
2. Lux handles mix mapping automatically when it comes to dynamic weather or diffuse scattering but it does not blend the base texture sets (as it does not know which textures you actually use and how you map these). So you will have to do this manually in your custom surface shader code – but it should be straightforward.

Doing it brute force for all members of the output structure could look like this:

```
o.Albedo = lerp(color1, color2, lux.mixmapValue.y);
o.Normal = UnpackNormal(
    lerp( tex2D(_BumpMap, lux.finalUV.xy), tex2D(_DetailNormalMap,
lux.finalUV.zw),
    lux.mixmapValue.y ) );
o.Specular = lerp(spec1, spec2, lux.mixmapValue.y);
o.Smoothness = lerp(smoothness1, smoothness2, lux.mixmapValue.y);
o.Occlusion = lerp(occlusion1, occlusion2, lux.mixmapValue.y);
o.Emission = lerp(emission, emission2, lux.mixmapValue.y);
```

Additionally you will have to enable mix mapping by declaring the corresponding **keyword** which would look like this:

```
#define GEOM_TYPE_BRANCH_DETAIL
```

Here we use a speed tree keyword in order to save keywords: Not easy to read but good in case you have a bunch of custom shaders or image effects in your project.

By default Lux assumes mix mapping being controlled by a value which is “constant” during the processing of each single fragment.

If you chose to use a texture input instead, the shader might adjust the mix mapping value within the loop of the Parallax Occlusion Mapping function giving you much more stable blending results. Enable “built in” texture driven mix mapping by adding a second keyword:

```
#define GEOM_TYPE_LEAF
```

Once the keywords are defined and the `mixmapValue` is set up all following Lux functions will take Mix Mapping into account when it comes to e.g. water accumulation or diffuse scattering.

Mix mapping and PM/POM

In case you use Lux Parallax or Parallax Occlusion Mapping and enable “built in” texture driven Mix Mapping you do not have to set up the `lux.mixmapValue` manually, as this is done by the `LUX_PARALLAX` function.

In this case the PM and POM functions expect the `_ParallaxMap` to contain:

- the height for the base texture set in the green color channel,
- the height for second texture set in the alpha channel,
- the mix map in the blue color channel.

The mix map will always be sampled using the base uvs. You may however define a Parallax Tiling relative to the base uvs.

Adding Parallax or Parallax Occlusion Mapping

Include the Lux PM and POM functions which will also add the needed inputs automatically:

```
#include "../Lux Core/Lux Features/LuxParallax.cginc"
```

Add the needed **properties**:

```
[NoScaleOffset] _ParallaxMap ("Height Map (G)", 2D) = "black" {}
_Parallax ("Height Scale", Range (0.005, 0.1)) = 0.02
[Toggle(EFFECT_BUMP)] _EnablePOM("Enable POM", Float) = 0.0
_LinearSteps("Linear Steps", Range(4, 40.0)) = 20
```

As the toggle “_EnablePOM” lets you switch between simple parallax mapping and parallax occlusion mapping we have to make multiple shader variants. Do so by **declaring a shader feature** (using another speed tree shader keyword):

```
#pragma shader_feature _ EFFECT_BUMP
```

`EFFECT_BUMP` enabled will enable POM, if it is disabled the shader will use simple parallax mapping.

Calling the Lux parallax function should be done before calling dynamic weather. It hasn’t any input parameters so it simply looks like this:

```
LUX_PARALLAX
```

It returns the offsetted uvs in `lux.finalUV` which you should use from here on.

Please note: `lux.finalUV` is a float4. `LUX_PARALLAX` also writes to `lux.height` and calculates the final `lux.mixmapValue`.

Parallax or Parallax Occlusion Mapping and Alpha Testing

Please note: In case you use Parallax Mapping or POM together with alpha testing and forward rendering things unfortunately get a bit more complicated as we have to specify a matching shadow caster pass which writes correct depth and samples shadows according to the extrusion.

Please have a look at the provided example shader: “Standard FullFeatures DoubleSided Cutout” to find out the details.

As you will see it is pretty simple to do as Lux provides all needed functions :-)

Adding Diffuse Scattering

Include the Lux diffuse scattering function which will also add the needed inputs automatically:

```
#include "../Lux Core/Lux Features/LuxDiffuseScattering.cginc"
```

Add the needed **properties**:

```
_DiffuseScatteringCol("Diffuse Scattering Color", Color) = (0,0,0,0)
_DiffuseScatteringBias("Scatter Bias", Range(0.0, 0.5)) = 0.0
_DiffuseScatteringContraction("Scatter Contraction", Range(1.0, 10.0)) = 8.0
```

In case you use mix mapping also add:

```
_DiffuseScatteringCol2("Diffuse Scattering Color2", Color) = (0,0,0,0)
_DiffuseScatteringBias2("Scatter Bias", Range(0.0, 0.5)) = 0.0
_DiffuseScatteringContraction2("Scatter Contraction", Range(1.0, 10.0)) = 8.0
```

Finally call the **scattering function** at the very end of the surface function.

You have to pass the albedo and tangent space normal. In case you do not have a tangent space normal simply pass half3(0,0,1)

```
LUX_DIFFUSESCATTERING(o.Albedo, o.Normal, IN.viewDir)
```

Diffuse Scattering does not need any **keywords** to be declared: You either call it or not.

But it can be disabled on a per material basis by changing the alpha value of the `_DiffuseScatteringCol` and (in case mix mapping is enabled) `_DiffuseScatteringCol2` to 0.0. All other alpha values (> 0.0) do not have any influence. Snow scattering however is always applied if snow is enabled.

Diffuse Scattering does not need the Lux fragment structure to be declared – unless you use other Lux features which depend on it.

Adding Specular Anti Aliasing

Include the Lux specular anti aliasing function:

```
#include "../Lux Core/Lux Features/LuxSpecularAntiAliasing.cginc"
```

Make sure your **input structure** contains:

```
float3 worldNormal;
INTERNAL_DATA
```

In order to make the shader be compiled out properly you **have to write to o.Normal** in the surface function. In case you do not sample any Normal Map simply write:

```
o.Normal = half3(0,0,1);
```

Finally call the **anti aliasing function** at the very end of the surface function:

```
LUX_SPECULARANITALIASING
```

Please make sure that Specular Anti Aliasing is globally enabled by editing the "**Lux Config.cginc**".

Adding Tessellation

Adding tessellation more or less is straightforward as Unity provides everything we need when writing surface shaders. See:

<http://docs.unity3d.com/Manual/SL-SurfaceShaderTessellation.html>

for further details.

But it gets a bit cumbersome when adding Lux surface features as Unity does not allow us to declare a custom **Surface Shader Input Structure** when using tessellation.

So we have to map our vertex output values to the default structure (see:

<http://docs.unity3d.com/460/Documentation/Manual/SL-SurfaceShaders.html>) or shift some calculations from the vertex to the fragment shader.

Please have a look at the provided “Standard Tessellation DynamicWeather” shader to find out how it deals with that limitations.

Currently Lux tessellation assumes that you want to tessellate and displace the given geometry. So tessellation always has to be used combined with parallax.

Please note: Unlike other custom surface shaders all shaders using tessellation have to use the standard “uv_MainTex” input.

Include the Lux parallax and tessellation functions which will also add the needed inputs automatically:

```
#include "../Lux Core/Lux Features/LuxParallax.cginc"
#include "../Lux Core/Lux Features/LuxTessellation.cginc"
```

Add the needed **properties**:

```
[NoScaleOffset] _ParallaxMap ("Height (G) (Mix Mapping: Height2 (A) Mix Map (B))
PuddleMask (R)", 2D) = "white" {}
// As we can't access MainTex_ST (Tiling) in surface shaders
_ParallaxToBaseRatio ("MainTex Tiling", Float) = 1
_ParallaxTiling ("Parallax Tiling", Float) = 1
_Parallax ("Height Scale", Range (0.0, 1.0)) = 0.02
_EdgeLength ("Edge Length Limit", Range(1, 40.0)) = 5
_MinDist ("Near Distance", float) = 7
_MaxDist ("Far Distance", float) = 25
_Phong ("Phong Smoothing", Range(0, 20.0)) = 1
```

Enable tessellation by declaring the corresponding **keyword**:

```
#define TESSELLATION_ON
```

Add the tessellation function to your **initial pragma directive**:

```
#pragma surface surf LuxStandardSpecular addshadow fullforwardshadows
vertex:LuxTessellationDisplaceMixMapped nolightmap tessellate:LuxTessEdge
tessphong:_Phong
```

Currently available vertex function are:

LuxTessellationDisplace – which simply displaces the vertices.

LuxTessellationDisplaceMixMapped – which displaces the vertices taking Mix Mapping into account.

tessellate:LuxTessEdge is a slightly modified Edge based Tessellation function which takes distance and the view frustum into account whereas **tessphong:_Phong** is just the built in function.

The provided sample shaders also use **nolightmap** in order to keep the amount of vertex data small. In case you need lightmaps or GI get rid of this and make sure that your appdata struct includes texcoord1 and texcoord2.

Declare the needed or desired **appdata** struct which has to be done before the includes:

```
struct appdata {
float4 vertex : POSITION;
float4 tangent : TANGENT;
```

```

float3 normal : NORMAL;
float2 texcoord : TEXCOORD0;
// Minimal struct, so lightmapping is not supported in this example
// float4 texcoord1 : TEXCOORD1;
// float2 texcoord2 : TEXCOORD2;
fixed4 color : COLOR0;
};

```

In case you use Mix Mapping you will have to make sure you call LUX_PARALLAX in the surface function.

The Tessellation Properties are named after the parallax ones but of course do include some unique properties.

MainTex Tiling: As we can't access MainTex_ST in custom surface shaders we have to specify the main tiling here to make the parallax map being sampled in the vertex shader accordingly. Simply enter the tiling of the MainTex ("Base (RGB)") here.

Parallax Tiling: Tiling relative to the MainTex Tiling value.

Height Scale: Determines how far the vertices will be offsetted.

Edge Length Limit: Tessellation level computed based on triangle edge length on screen – the longer the edge, the larger the tessellation factor will be.

Near Distance: Full tessellation at this distance and below.

Far Distance: No tessellation beyond this distance.

Phong Smoothing: Lets you smooth low-poly models.

Tessellation: The tessellation factor.

Please note: As tessellation does not allow us to optimize the water flow and water ripple calculation based on the globally defined "Detail distance" you should consider swapping the tessellated material for e.g. a material using simple parallax instead on lower LOD levels.

Right now tessellation is only supported in custom surface shaders.

Hierarchy within the surface function

In order to make anything work properly you will have to call the Lux functions in a certain order within your surface function:

1. Initialize the Lux Fragment structure
This structure will hold all needed values which can be accessed by the included functions or by your custom code.
2. Get and set the Mix Mapping Value if needed.
3. Call Lux Parallax functions
This will return the extruded UVs, the height for the given fragment needed by wetness accumulation and may calculate the final mix mapping value.

4. Initialize Dynamic Weather

This will calculate the amount of accumulated water and snow and returns refracted UVs in `lux.finalUV` in case there are any rain ripples or water flow.

5. Do your regular stuff

As now we have the extruded and refracted UVs you can do the albedo, normal and specular look ups. Use `lux.finalUV` to sample the textures.

6. In case your custom stuff uses the metallic workflow make sure you convert all outputs to the specular workflow calling: `LUX_METALLIC_TO_SPECULAR`

7. Apply Dynamic Weather

This lerps all outputs towards water or snow

8. Apply Diffuse Scattering

Using Lux Lighting features and functions

Area lights

Area lights will be supported automatically in case you include the Lux lighting functions and make the shader multi_compile using:

```
#if defined (UNITY_PASS_FORWARDBASE) || defined(UNITY_PASS_FORWARDADD)
    #pragma multi_compile __ LUX_AREALIGHTS
#endif
```

The multi_compile directive is only needed in forward rendering. Deferred shaders always support area lights if these are enabled by script.

Please have a look at the [dedicated example](#) above.

They are supported in deferred shading even on any other not customized shader.

Diffuse fill lights

Diffuse fill lights will be supported automatically in case you include the Lux lighting functions. They are supported in deferred shading even on any other not customized shader.

Lambert lighting

Lambert lighting is only interesting when it comes to deferred shading. It is supported on all shaders without the need for any tweaks as soon as you assign the “Lux Internal-DeferredShading” shader.

Anisotropic lighting

Creating a custom shader using Lux anisotropic lighting will need the following steps:

1. Create a new shader in the folder “Lux custom Anisotropic Shaders”.
2. Declare the Lux anisotropic lighting function inside the #pragma surface directive.
3. **Include** the LuxAnisoPBSLighting.cginc.
4. Adjust the **surface output structure**.
5. Add the needed **properties** by e.g. copy & paste the needed properties from the “LuxShaderProperties.txt” file.

Please have a look at the “Lux Anisotropic” shader to find out about the details.

Please note: You will always have to write to `o.Normal` to make the shader compile out the needed `tangentToWorld Matrix`.

Please note: Smoothness should never go up to 1.0.

Please note: Deferred shading does not handle Metallic = 0.0 correctly (as it would cost some extra instructions). So you have to take care about this when authoring your metallic texture or setting up the metallic value using a slider.

Advanced anisotropic lighting

The anisotropic BRDF uses world tangent direction, roughness along tangent and roughness along bitangent as inputs which are fed from simply sliders in the provided sample shaders. By sampling the needed values from additional textures instead you will be able to create much more flexible materials like materials which support anisotropic reflections on some parts and isotropic reflections on others (where roughness along tangent = roughness along bitangent).

Skin shading

Lux skin shading is based on pre-integrated lighting and combines deep subsurface scattering from a static depth map (like translucent lighting) and light scattering calculated dynamically based on the given curvature of the (skinned) surface.

Subsurface color, scattering power, distortion and scale are controlled globally by the “Lux Setup” script.

Skin lighting may fade out over distance. So it gets pretty easy for you to create lower LODs of your characters using a combined material for e.g. skin and cloth and standard lighting and swap between LODs without any harsh visible “jump”.

Light fading is controlled globally by the “Lux Setup” script where you can enable/disable it as well as setting up the range and fade distance.

The default skin shader supports **Micro bumps** – which add very fine details to the specular highlights. Enable this feature only if you have cut scenes and/or very close distances between camera and skin as in most other cases it just will not be visible thus being a waste of resources.

As skin lighting is not a trivial thing i do not go into details here writing your own skin shader but you may e.g. consider using a “static” curvature texture instead of having curvature being calculated in the shader dynamically.

Please have a look at the provided skin shader to find out more.

General Texture Import Settings

Lux uses a bunch of combined texture like the global snow mask or combined height and mask textures when it comes to mix mapped parallax occlusion mapping.

These texture do not contain regular “color” information, so you will have to make sure that Unity bypasses sRGB sampling. Do so by switching the texture import settings to “Advanced”, then check “bypass sRGB Sampling”.

When using DXT5 compression using more and more channels will most likely create more and more artifacts. So if you e.g. use Parallax and Mix Mapping you will have to store both heights in the green and the alpha channel. The mix map might be stored in the blue channel and the puddle mask might go into the red channel. So if all channels are used your

heightmaps might suffer from this. Consider setting one channel (blue or red – depending on what feature you may lift e.g. to vertex colors) to black – or think about using an uncompressed texture...

Custom example Surface Shaders

Lux custom Standard Shaders

Terrain Shader

Simple example of a custom terrain shader “family” including first pass, add pass and base shaders that supports a limited feature set of “Dynamic Weather”.

In order to set up Dynamic Weather and adjust parameters like Snow Mask Tiling you will have to edit the terrain material.

Standard WavingGrass

Overwrites the built in Grass shader for terrain (does not support billboarded grass though) and allows you to render grass in deferred using lambert lighting in order to get rid of strange specular reflections at grazing angles.

It also supports dynamic snow using a simple custom snow accumulation function.

Simple Metallic

Super simple shader using the metallic workflow within the custom surface shader block.

It shows how and where to apply the `LUX_METALLIC_TO_SPECULAR` macro to make sure that all outputs are mapped properly.



FullFeatures CustomSnowMask

Shader which shows how to add custom texture driven snow masks and extra snow normals to create e.g. skidmarks.

FullFeatures DoubleSided Cutout

This shader is the most complex one as it includes all surface features and supports alpha cutout as well as single sided geometry.

It shows how and where to apply **VFACE** in order to get correct shading on backfaces and comes with a custom **example shadow caster pass** to make all things work properly when it comes to forward rendering.

Tessellation DynamicWeather

A simple shader supporting displaced Tessellation and Dynamic Weather.

Tessellation DynamicWeather MixMapping

A simple shader supporting displaced Tessellation, Dynamic Weather and Mix Mapping.

The Tessellation Properties are named after the parallax ones but of course do include some unique properties.

MainTex Tiling: As we can't access MainTex_ST in custom surface shaders we have to specify the main tiling here to make the parallax map being sampled in the vertex shader accordingly. Simply enter the tiling of the MainTex ("Base (RGB)") here.

Parallax Tiling: Tiling relative to the MainTex Tiling value.

Height Scale: Determines how far the vertices will be offsetted.

Edge Length Limit: Tessellation level computed based on triangle edge length on screen – the longer the edge, the larger the tessellation factor will be.

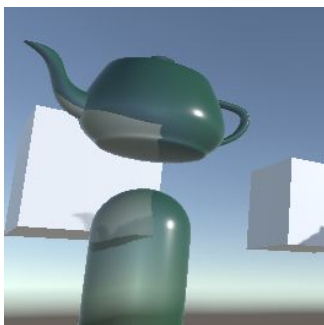
Near Distance: Full tessellation at this distance and below.

Far Distance: No tessellation beyond this distance.

Phong Smoothing: Lets you smooth low-poly models.

Please note: As tessellation does not allow us to optimize the water flow and water ripple calculation based on the globally defined "Detail distance" you should consider swapping the tessellated material for e.g. a material using simple parallax instead on lower LOD levels.

Right now tessellation is only supported in custom surface shaders.



Standard Refraction Geometry

This shader allows you to create transparent materials which refract the background based on the surface normal (geometry) as well as on the normal from texture input.

Refraction: Overall amount of refraction.

- **Geometry:** Influence of the geometry's normal on refraction.

- **Normalmap:** Influence of the normal map on refraction.

Fresnel: Lets you control the ratio between refraction and reflection which in this case is the albedo color.

Please note: refractive shaders will not refract other refractive materials by default as they use the same grab texture which simply does not contain refractive materials.

Lux custom Translucent Shaders

Translucent Base

Most simple custom shader using translucent lighting.

Translucent DynamicWeather

Simple translucent shader supporting Dynamic Weather. As the shader does not sample a height map, lux.height is set from Normal.z. to get some variation as far as the water accumulation is concerned.

Translucent WavingGrass

Overwrites the built in Grass shader for terrain (does not support billboarded grass though) and allows you to render grass in deferred using lambert lighting in order to get rid of strange specular reflections at grazing angles. It even adds simple translucent lighting.

It also supports dynamic snow using a simple custom snow accumulation function.

As the terrain grass does not have a specific material you will have to edit the shader in case you want to tweak translucent lighting.

Shader needs Lux Plus.

Lux custom Anisotropic Shaders

Anisotropic Base

A simple shader that supports anisotropic lighting and lets you define or tweak the tangent direction using a texture input and/or slider values.

As all anisotropic shaders support translucent lighting you may enable it – even in this simple example.

Lux Plus only: Deferred dithering

Enable Dithering: As—when using deferred anisotropic lighting on super smooth surfaces—specular highlights might suffer from banding artifacts at very close viewing distances you may enable dithering. Doing so will slightly vary the world tangent direction which gets written into the gbuffer.

Animate Dithering: In case you use temporal anti aliasing you may activate this option. Please note that temporal anti aliasing will some kind of recover the banding so you might have to raise the *Spread* value.

Spread: Determines the amount the world tangent will be varied from pixel to pixel. 0.0 equals no variation, 0.1 equals a lot of variation. You most likely want to keep this value as close as possible to 0.0.

Start Distance: As banding artifacts are not visible at far viewing distances you may specify a distance at which dithering jumps in. When rendering objects which are beyond the start distance the shader will skip dithering.

Fade Range: Range over which the dithering fades in starting at the *Start Distance*.

Anisotropic Hair

This shader uses anisotropic and translucent lighting to “simulate” hair shading. It is not a “real” hair shader supporting multiple specular highlights but it will give you pretty nice results within the limited range of available lighting models when it comes to deferred shading. It supports single sided geometry, handles backfaces correctly and allows you to use dithered opacity.

Hair shader specific Properties

Enable dithered Opacity: If checked the shader will not use simple cut out transparency but dithered opacity revealing the half tones of the alpha mask. Combined with temporal AA this might give you some kind “alpha blended” opacity.

Please note: The shadow caster pass still uses alpha cutout – so the “_Cutoff” property determines the shape of the shadow caster.

Enable Metallic Gloss Occlusion Map: If checked Metallic, Smoothness and Occlusion are taken from texture input multiplied by the slider values for Smoothness and Metallic.

Vertex Color Occlusion: Gets added on top of the “micro” occlusion baked into the Metallic Occlusion Smoothness texture.

As you will probably use only a few different textures packed into an atlas using vertex colors on top will give you much more lively lighting as it acts as some kind of “macro” occlusion.

Vertex color occlusion is taken from vertex color green.

Tangent (RG): Tangent direction texture tweaking the final tangent direction on a per pixel basis.

Base Tangent Direction (XYZ): A Value of 0.0 / 1.0 / 0.0 matches the original tangent direction whereas a value of 1.0 / 0.0 / 0.0 would describe the bitangent direction.

Strength: A value of 1.0 would make the shader use the original tangent dir (tweaked by Tangent Direction texture input, see below) whereas a value of 0.0 would make the shader use the specified Base Tangent Direction.

[Anisotropic Lighting Details >](#)

Hair specific Lighting

Enable Translucent Lighting: If checked the shader will add translucent lighting. Distortion, Power, Scale and Shadow Strength are globally controlled by script. The translucency mask is derived from the metallic mask.

Diffuse Scattering: Lets you define a diffuse scattering color. If set to black the shader skips diffuse scattering.

Scatter Bias: Lets you raise the diffuse light scattering by adding a constant.

Scatter Contraction: Lets you sharpen the diffuse light scattering towards grazing angles.

Please note: Adding diffuse scattering when using the metallic workflow is not really correct as it will change the finally calculated specular color as well. But it looks ok :-).

Lux custom deferred decal shaders

Deferred geometry based decals let you add small details or decals which are smoothly blended towards the underlying surface – just like you would get in forward when using alpha blended shaders.

Deferred geometry based decals do not work like the deferred decals in Unity's gbuffer examples: Deferred geometry decals need you to actually place decal geometry which fits the underlying surface. But they allow you to smoothly blend between decal and the surface behind – as otherwise only supported in case you use alpha blending and forward rendering.

Blending is done directly in the gbuffer which unfortunately needs 2 shader passes in order to write all parameters needed by physically based rendering properly.

Deferred geometry decals support all lighting features including ambient lighting and ambient specular reflections.

Please note: Currently deferred geometry decals must only be placed on top of geometry using standard lighting.

Please note: Adding deferred geometry decals on top of deferred geometry decals will most likely cause z-fighting issues.

Please note: Adding deferred geometry decals on top of highly displaced surfaces will most likely make the decals float.

So after all these “please notes” what are deferred geometry decals good for in the end? Especially if you could use Mix Mapping to seamlessly blend between 2 independent texture sets?

The answer to this: In case you have only sparse or even dynamically occurring details here and there which shall show up using different features or textures deferred geometry decals might be much faster to render and would give you more freedom than Mix Mapping.

Please note: In case you are using forward rendering deferred geometry decals are neither supported nor needed: Simply use an alpha blending shader instead.

Standard DeferredDecal Parallax

A simple decal shader that supports parallax and parallax occlusion mapping.

Standard DeferredDecal FullFeatures

A more advanced shader which supports parallax and parallax occlusion mapping as well as Dynamic Weather.