



V1.1
Documentation
January 26th 2016

Preface

Thank you for buying UBER – Standard Shader Ultra. As the name suggests it's based on Unity's Standard Shader principles and uses standard PBR lighting implementation. On the top of the Standard Shader I've added a lot of new features that can enhance your visual output dramatically.

Mainly they are based on very advanced POM (Parallax Occlusion Mapping) techniques that give impression of extruded surfaces without necessity to use complex geometries or tessellation. Although tessellation is also implemented (displacement & Phong smoothing), your output platform doesn't need to handle it. For some class of objects it's even better to use parallax occlusion mapping instead of tessellation. Together with enhanced detail mapping UBER will provide you with fast approximate translucency model, dynamic snow with glitter, dynamic water that flows down the slopes of your mesh and more. Every shader included has set of "core" functionalities and the difference between shaders are basically based on their tessellation/parallax technique and optional refraction.

Shaders are configurable for their vertex color usage which can control many aspects of your model from detail/snow/water masks thru AO baked or colorization of diffuse output. Not only shaders are provided by the package but also a lot of other helpful systems like global weather script can influence all scene objects at once with only a few physical parameters like fall intensity, temperature and the script will adjust water/snow values on the objects for you.

Installation of package is very straightforward. After importing it you'll see 2 subpackages inside – install them in the provided order. Disable MSAA (GPU antialiasing) in Unity quality settings to not break HDR rendering in forward lighting mode (example scene might not looking good then). Change your color model to Linear for best PBR experience as well.



Textures and example models included rely on the importers that should be present in your project. That's why you need to import shaders and examples after scripts that are placed in first subpackage.

Tomasz Stobierski

Table of contents

1. Basics and UBER core functionality

Main Maps

Secondary Maps

Wetness

Ripples – flowing water

Rain – animated droplets

Translucency

Glitter

Snow

Global params script controller

Presets

2. POM & Tessellation

Core Parallax Occlusion Mapping features

Extrusion Map

Self-shadowing

Distance Map

POM Advanced

Tessellation

3. Triplanar, 2 Layers & additional stuff

Triplanar selective

2 Layers

Edge emission with cutoff bleeding mode

2 Sided variants

Refraction shader variants

Texture channel mixer

4. Customization

1. Basics and UBER core functionality

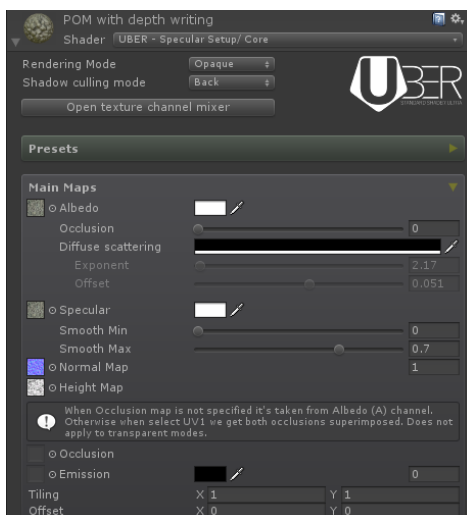
After installation you can go to example scene and look around there are a few objects that have UBER shader used and properly setup for rendering. Let's start with the basic functionalities that I'll describe section by section. They are depicted in colored boxes in shader material inspector. Some of them are optional in the shader and you can select the feature by checkbox at the left side of section. Examples are – **POM, Snow, Water, Translucency, Glitter**. Some sections are persistent and available in all shaders like **Presets, Main Maps**. Other sections are shader variant dependent (for example parameters specific to refraction, tessellation).



Looking at all possible sliders and controls available might be confusing. You can fold/unfold section using small triangle - dropdown like button on the right side of section header. This helps keeping visible only the controls that you're currently working with.

Main Maps

Let's consider material that's attached to **POM Box Z-write** game object example. It has Core shader selected from **UBER – Specular Setup** shader dropdown folder.



As we see it's very similar to what you're familiar with – Unity Standard Shader. I've selected **Albedo** map which stores the base color of the object in (RGB) channels. The difference is Ambient Occlusion storage. Instead of opacity, (A) channel of the albedo texture in UBER holds Ambient Occlusion which is controlled by **Occlusion** slider. That's the case when **Occlusion map** texture seen at the bottom of the section is empty. If it's not empty occlusion is taken from this dedicated map. The same for transparent rendering modes (topmost dropdown in inspector). Transparent modes store opacity in Alpha channel of Albedo texture so it would interfere and occlusion is taken from dedicated **Occlusion map** texture then as well.

In this case (A) channel of the texture attached to **Albedo** doesn't have occlusion data so I set the occlusion slider value to 0.



For Opaque and Cutout objects UBER can hold Ambient Occlusion data in (A) channel of **Albedo** map. As soon as you select **Occlusion** map texture, AO is taken from this dedicated map instead of **Albedo** (A) channel.

Color selector next to the **Albedo** map is always visible and is always used. (RGB) of the color is multiplied by Albedo texture (RGB) - you can always use albedo texture color tinting. Color picker is HDR, so in case your albedo texture is dark you can boost it to bright value with HDR tint. Keep tint color white to not influence albedo texture. In case of transparent rendering modes (A) value of color multiplies transparency value taken from texture (A) channel.

Diffuse scattering color can boost albedo color output visible at Fresnel/grazing view angles. It's also known as "peach fuzz effect". Useful for plenty of applications like velvet fabrics or gentle dust on the surface. Keep the color (RGB) black to turn off the feature. When (A) of the scattering color is zero UBER will multiply albedo color by diffuse scattering color (x4 for higher range). When your albedo is

black it would do nothing (as tinting black color does nothing). That's the case you can use (A) of scattering color. When it's 1 we actually don't multiply color but use scattering color (RGB) instead (x4). **Exponent** and **offset** sliders control falloff of the effect.

Specular map texture is like you know it in Unity's Standard Shader. (RGB) is specular color while (A) is smoothness (gloss). For metallic setup R channel would control metalness.



Specular color (metalness slider in metallic setup) is always used and works as "PBR tint". You need to remember setting spec color tint to white to get unaffected value from texture. The same for metallic setup – set metalness, smoothness to 1 to take unaffected values from specified metalness/gloss texture.



When your spec/metal + smoothness texture has filename with `_Gloss` suffix it will be automatically processed by UBER texture importer. It improves equalization of specular energy so that average amount of specular energy (brightness) stays the same for subsequent MIP level of smoothness map. Additionally if your normalmap name has the same prefix as spec/gloss map but has `_Normal` suffix UBER will try to incorporate variance into MIP smoothness levels. It can improve the look of shiny surfaces that have small hi-frequency details on it. Without filtering with variance you could experience either "huge mirrors" effect far away like described in Black-Ops tech PBR paper or noisy jittering upclose. Remember to make normalmaps read/write enabled before reimporting spec/gloss maps before. After import and baking variance into smoothness data you can set back read/write enabled flag to none.

New to UBER is very handy ranging feature - **Smooth min/Smooth max** sliders. When you keep it default (0, 1) smoothness works like in Unity. But often your smoothness is not calibrated the way you like. Correcting this would mean work again in imaging software which is time consuming - going back and forth to Photoshop/Unity to realize that most glossy parts of the texture are too glossy or most rough parts are too rough... With the sliders you can range output smoothness. Where your gloss value from texture is 0 – value of **Smooth min** will be output. Where your gloss from texture is 1 – **Smooth max** is taken. Values in-between are interpolated. So – if your smoothness map is normalized (0..1) – you can simply select range on the sliders to 0.2 – 0.7 and output gloss will be mapped to this range (most rough parts will have 0.2 smoothness output, most glossy parts will have 0.7 output).



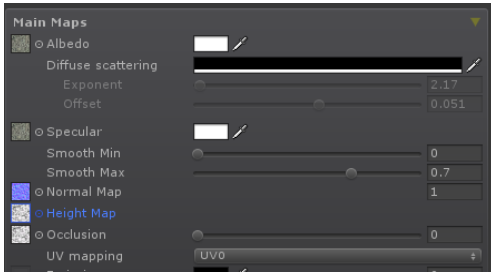
You can even invert gloss range - select 1, 0 for sliders (instead of 0, 1) and your most glossy parts will become most rough and vice versa.

Normal map is self-explaining and works like in Standard Shader.

Height map is used for parallax displacement effects and for wet effects. When POM (Parallax Occlusion Mapping) feature is disabled – default PM (Parallax Mapping) is used and slider on the right side of map is present to control parallax displacement. In above case (**POM Box Z-write** game object example) POM is turned on and displacement amount is controlled separately in POM section described in the next chapter.

Occlusion map is used like in Standard Shader – when it's filled by texture, its G color value is used to control AO effect. This map stores also mask for translucency and glitter effect (B). For 2 layers shader variant all layers are used (RG for 1st layer, BA for 2nd to control AO and translucency/glitter). Look at ear game object example – map with AO and translucency mask (with veins baked) is used.

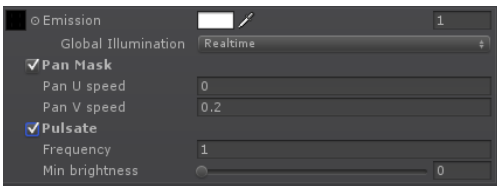
For ambient occlusion we can also use secondary mask. It's the case when UV coords are selected to be UV1.



Here **Occlusion** map is used – we don't have occlusion slider available below **Albedo** map as occlusion strength is controlled by the dedicated map. The slider is placed on the right side of Occlusion texture now. Below is **UV mapping** dropdown. By default it's set to UV0. Sometimes you'd like to have AO baked in atlas per scene to get AO based not on the detail texture but on scene geometry. Instead of using postFX AO (which would just darken everything rather than damping indirect light) you

could have AO baked for scene in atlas with UVs stored in 2nd set for each mesh. When you select UV1 from **UV mapping** dropdown above you'll have access to both – AO stored in (A) channel of Albedo map (slider will appear back there) and AO stored in Occlusion map. Both occlusion values will be superimposed (multiplied).

Emission map can be selected to control emissiveness of the object on its parts rather than as a whole (when emission color is not black and emission map is empty).



When emission map is present we can select two other options to animate emission – pan its mask (stored in A channel of emission texture) or to pulsate the emission color. Look at **Animated Emission** example game object.

Secondary Maps

The section is meant to introduce detail maps that are mixed with primary maps (with different tiling). UBER introduces PBR values for secondary maps which is not present in Standard Shader. This way you can control detail smoothness and specularity, not only color and normals.



Secondary maps / details are meant to work in 3 modes:

1. Simple - when you provide **Detail Mask** texture only. Useful for tinting some parts (when tint color Alpha next to **Detail Albedo** map is set to 1) and influencing PBR (when **Detail PBR** slider is set to 1)
2. Textured - when **Normal Map** or **Detail Albedo** texture is provided (or both)
3. Textured with specular map - like above but with specified detail **Specular** (Metalic in case of the other setup) map.

By default detail normals are blended with main normals. If you'd like detail normals to override main normals – use **Detail normal lerp** slider.

When **Detail Mask** is provided (R channel of the texture is taken) details will appear only on masked parts. The same for R channel of vertex color of the mesh. When it's not set (vertex color buffer not present in mesh) it's considered to be 1 (white/full) and you get no vertex based masking.



You can paint detail presence with any vertex color painter – R channel. Using Detail Mask you can additionally enhance resolution with some noise texture to hide lack of polygon tessellation on your object.



Detail Albedo color is tint for detail color (when texture is present). (A) value is opacity. So – as soon as you select a detail map, like **Normal Map** your object will not look right. By default (A) of color tint is 1 (detail color opacity is full) and RGB values are used. Turn color (A) to 0 so detail does not influence object color. The second is **Detail PBR** slider set to 1 which means specular (metalness) and glossiness value override main PBR values. Set the slider to 0 to bring PBR values back to main (primary) ones.

Even without specular/gloss map specified there might be situations where PBR of detail might be useful. For example you have some “water puddles” stored in detail mask (use vertex color R to optionally distribute puddles over the object surface). Then you can set Detail PBR slider to 1 and use specular RGB dark grey (0.1 actually for correct water spec color) with high gloss (detail PBR color (A) channel set to 1). Then your puddles of water will be shiny. Use flat detail normalmap and set **Detail normal lerp** to 1. Puddles will be flat (as water surface shouldn't have main normalmap from underlying surface). This way you can make your object partially wet even without using dynamic water features described later.

Wetness

This section controls dynamic water effects. Inspector state is taken from **Sphere Flow** game object in the example scene. Level of water coverage can be controlled by 3 factors – vertex color (A) channel, Detail Mask (R channel – look at secondary maps above) when detail mask is present and **Water Level/Const Wet**. With vertex color you can define where water is present on the model, useful when the model is partially placed under a shelter (bridge, roof, tree...). Detail mask (R) helps breaking water coverage controlled by vertex color when your model has larger triangles.



Two main water level controllers are **Water Level** and **Const Wet** sliders. First one is heightmap dependent – water will appear first on parts with darker (placed lower) heightmap values. This way water can appear between rocks. For maximum **Water Level** value even the topmost parts (white parts of heightmap) will be covered by water. **Const Wet** controls wetness on the surface that is independent on heightmap – it's just “wet”

instead of deeper water that can flow and receive rain droplets. So – when you'd like to increase glossiness of the surface and make it a bit darker only – use Const Wet slider.

The water itself has its own **Color/opacity** (RGB / A). **Wet darkening** controls how much the surface diffuse (albedo) color is darken when it's wet. For mineral materials like rocks, pavements we can observe this effect. For hard plastic or metals which present no surface porosity darkening will be zero. Wet smooth plastic would have the same color when it's wet, only glossiness/specularity increase. PBR for water is based on specular model, physically correct values are set by default – **Specular & Gloss** color are dark grey (RGB specular) with high glossiness (A). With dark specular color water exposes strong Fresnel effect – it's reflective only at grazing angles. When we look top-down the water is transparent (low A value of water color). When you'd like the water to hold it's own normal – set **Normal override** near maximum – 1. Specular light bounced by still water should look flat, not modified by underlying surface normals. The same for flow – we can see animated ripples that have

own normals of the water surface. You can set the slider lower if you'd need the water to look more like it's running on the surface rather than like accumulated water (puddles, flow between features caused by heightmap).

Ripples – flowing water

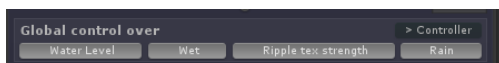
Ripple tex is selected by default so you don't need to provide UBER shader with it. Notice that this texture is shared with snow feature (when it's used) to save this kind of resource. This is normal map that is sampled twice with animated offset which gives the effect of flow. Slider next to the texture controls ripples normal strength. Texture is sampled in world space top planar (world XZ position) with specified **Tiling**. Setting this to 4 means – 4 ripple texture tiles across 1 world unit – larger values give smaller ripples. Top planar mapping makes water ripples continuous across adjoined meshes but also means that at steep slopes ripple texture will be stretched. **Gloss filtering** is useful when your flow ripples are tiny and seen at grazing (fresnel) angles. Water is glossy medium and environment reflection can be noisy. With filtering we use smaller gloss values at grazing angles. This smoothens high frequency artifacts on the water. **Anim speed** is obvious – controls how fast water flows down the mesh slope. With **Global timer** checkbox selected animation speed is controlled by global shader value passed by **UBER_GlobalParams.cs** script that's attached to the main camera in the example scene. It's useful when you'd like to manage water flow in the whole scene for sake of controlling weather conditions. When the temperature is below 0°C it would freeze and flow animation should be set to zero speed. Look at the bottom part of the water section in material inspector above. Subsection is named **Global control over** and has small [**>Controller**] button. Clicking this is the same as selecting **Main Camera & GlobalControl** game object in the example scene. In the inspector you can see this script:



When you uncheck **Simulate** (first in the **Rain/snowfall controller**), you can select 0 in **Flow Time scale** field. Water will stop flowing on the **Sphere Flow** game object (as far as it still has **Global timer** checkbox selected). So – with global timer we can control flowing animation speed which is **Flow Time Scale** multiplied by material **Anim speed** value.

The reason you had to uncheck **Simulate** is that global values for water (**Global Water & Rain**) and snow (**Global Snow**) are dynamically affected by parameters of **Rainfall/snowfall controller** part. With selected **Fall intensity** slider and positive **Temperature** flow timescale will be positive (water is “unfrozen” and can flow). When temperature goes below 0°C water freezes.

Let's back to the previous screenshot (**Wetness** section of UBER material inspector). On the bottom we've got this 2 state buttons in **Global control over** subsection.



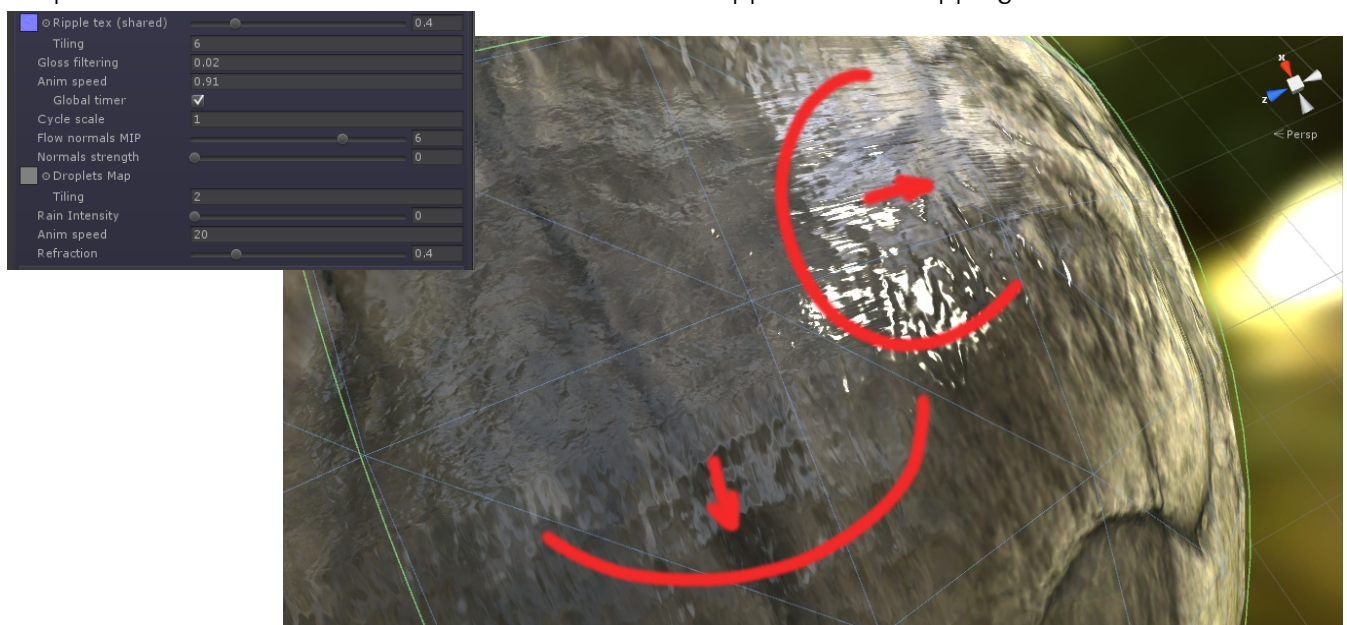
They are: [Water Level], [Wet], [Ripple tex strength], [Rain]. With these selected – global params script has control on such parameter pairs:

1. [Water Level] selected – material Wetness **Water Level** value is multiplied by **Water Level** value of global script
2. [Wet] selected – material Wetness **Const Wet** value is multiplied by **Wetness Amount** value of global script
3. [Ripple tex strength] selected – material Wetness **Ripple tex** slider value is multiplied by **Flow Bump Strength** value of global script
4. [Rain] selected – material Wetness **Rain intensity** value is multiplied by **Rain intensity** value of global script

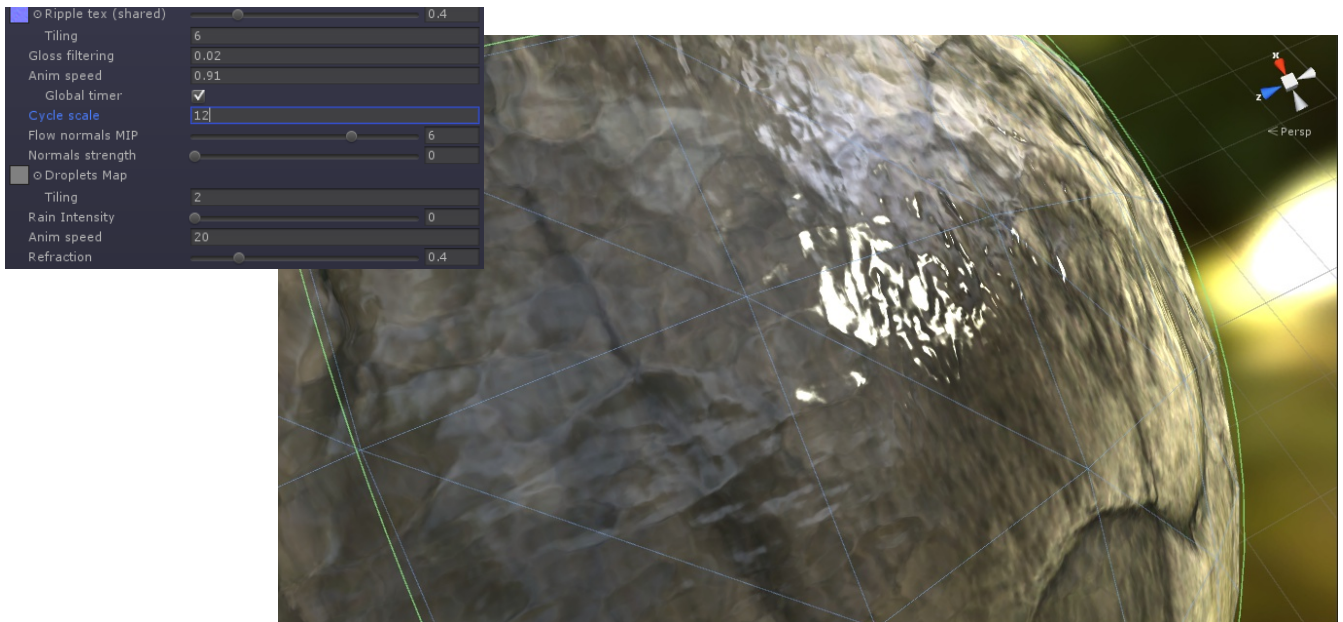
As mentioned – **Global timer** checkbox selected in material inspector pairs material **Anim speed** value with **Flow Time Scale** value in global script. Global Params controller will be additionally described with snow feature later.

Next numerical value in Wetness section of material inspector is **Cycle scale**. Higher values will remove stretching artifacts on flowing parts that have different slope. Constant slope example would be a plane – no matter how small cycle scale is we won't see any stretching artifacts. Changing slope example would be a top part of sphere or small hill. On the very top of the hill direction of flow is undefined because it's horizontal flat while each neighbour part of the peak will have different flow direction.

For the following screenshot I intentionally set cycle scale to 1 (instead of 12 value that's present on the example scene object) and increased anim speed to exaggerate stretching artifacts. Different parts of the sphere have different flow direction which leads to ripple texture mapping discontinuities.



Setting high **Cycle scale** value will remove it at the cost of shortening flow cycle duration:

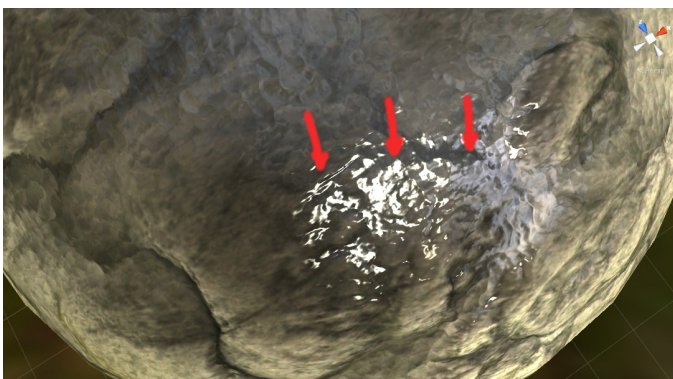


With such **Cycle scale** notice flow have tiling increased (larger ripples). So **Anim speed**, **Tiling** and **Cycle scale** are mutually dependant. With very high **Cycle scale** you actually won't see any flow but something that looks like quick pulsation, because repeating flow cycle duration will be very short. You should set the lowest **Cycle scale** value that is acceptable to reduce stretching artifacts for given **Anim speed**. Then adjust **Tiling** to the value needed. However – material in the example scene has values set to the realistic water behaviour and sizing that works fine as world unit is meant to be 1 meter.

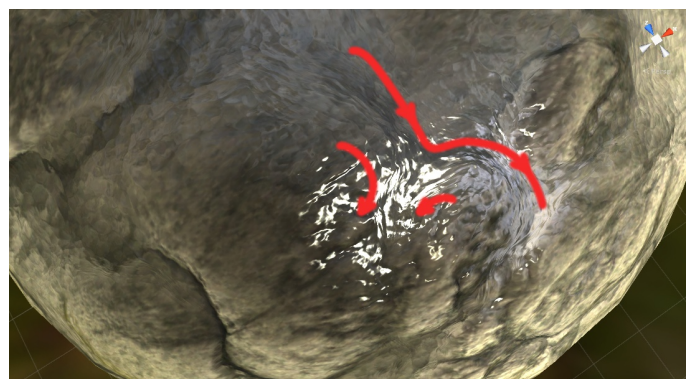


Set Ripple tex map to empty (None) to disable ripple flowing effect – usefull for still water puddles. Additionally this will gain a bit of performance as water flow is not free for GPU.

Flow normals MIP and **Normals strength** define the way slopes for flow direction are taken. For **Normals strength** set to 0 **Flow normals MIP** doesn't matter and flow direction is based on mesh world normals alone. With **Normals strength** set to higher value (in the example scene sphere it is set to max – 4) we take primary Normal map value as well to compute the slope for flow direction. **Flow normals MIP** is mipmap offset of primary normalmap taken for flow. In most cases primary normalmap has some tiny high frequency details and computing flow direction basing on such “micro detailed slope” would lead to unnatural look. However, when we take filtered, higher MIP levels of normalmap to determine the flow direction we can achieve a look that resembles water running “around” the surface features like stones on the example sphere object:



Normals strength set to 0



Normals strength = 4, Flow normals MIP = 5

Rain – animated droplets

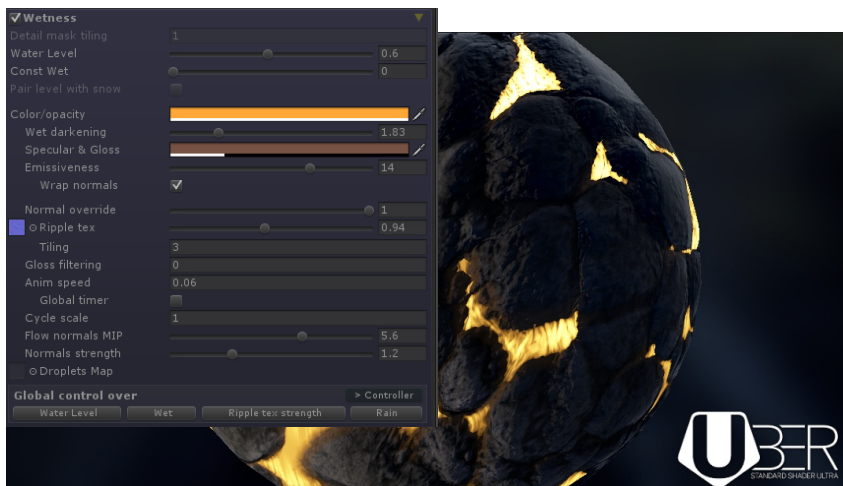
Separate feature of the UBER water is rain droplets animation. The **Droplets Map** texture is provided in the package and is set for material by default. **Tiling** control how big droplets will be (higher tiling means smaller droplets). **Rain intensity** means how “dense” and frequent splashing droplets will appear. **Anim speed** controls the movement of droplets.

By default vertex color channel B controls rain droplets coverage. This way you can exclude some wet parts from receiving droplets when they are sheltered.



Set Droplets Map texture to empty (None) to disable rain droplets effect when you'd like to use water w/o droplets. As the feature is not free for GPU load don't use it when it's not actually needed.

Refraction slider value is shared between Water flow (ripples) and rain and controls the level of refractive distortion of surface seen under the water. Setting refraction to 0 would look a bit unnatural – even if ripples would strongly influence output normals. However too strong refraction would distort underlying surface image to the level its details are unperceivable. For very rapid “stream like” flow it might be desirable though.



When you set **Emissiveness** of the dynamic water, you can simulate lava-like flow and **Wrap normals** checkbox appear then. Compare how emissiveness is implemented with and without this checkbox. For lava it looks natural with the switch selected. Set bright orange color for water and slow anim speed. This is how this is setup for example scene.

Translucency

This feature is meant for subsurface scattering effect. Although it's not physically based but approximate solution it can deliver fast and very pleasant visual effect.



The idea is based on DICE model presented on GDC 2011. Translucency transmission is provided for backfacing parts when we look at the object against the light (object is placed between camera and light source). Amount of translucency depends on the angle between light ray and camera to surface point ray.

This is how the feature looks like in UBER material inspector. **Color** defines light source tint. For most objects this is just white. In some cases it's tinted - assume ear example above – translucent color is red as blood below the skin influences the color – an ear observed against the light has strong red hue. Amount of translucency is controlled by **Strength** slider. **Constant** value is always added regardless of any light angle. So – for directional light it's always present. For point light it depends on falloff distance.

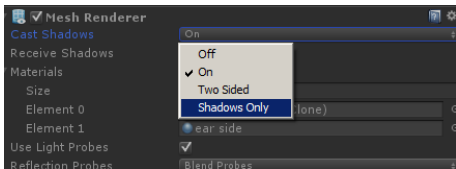


Occlusion mask is amount of masking taken from **Occlusion map** texture (R) for specular setup and (B) for metallic setup. You can deliver nice transmission maps from newest xNormal where translucency (transmission) maps can be baked. It's just mesh inverted ambient occlusion calculated with inverted normals. Such approximation of occlusion mask works surprisingly well as thin protruding mesh parts would deliver you dark AO value when we calculate them with inverted normals (pointing inside object). When we then inverted these dark parts they will be bright masking value for transmission and for example nose, hands on above statue model are more translucent than central parts of the model (belly).

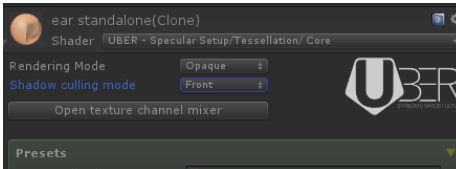
As object can cast shadows on itself you wouldn't see any translucency. That's why **Suppress shadows** slider is introduced. It gains translucency strength on parts that stand in shadow. Other trick might be using proxy solution, but it does work well for convex objects only (like spheres).

Such setup would be like this:

1. Copy your object
2. For first one set **Cast Shadows** dropdown in mesh renderer component to **Shadows only**



In top of the UBER material inspector set shadow culling to front:



Unity allows to set shadow casting to **Off** (shadow caster is not rendered), **On** (shadow caster culling Back), **Two Sided** (culling turned off so both sides of caster are rendered into shadowmap) or **Shadows Only** (which is the same as Back culling but actual object is not rendered, only shadow caster). With this additional

option in UBER material inspector you can ask Unity to render shadow caster faces that are oriented back to the light (polyes normally invisible from light perspective).

3. For 2nd object you actually render the model (turn off shadow casting for it).

This is not perfect as we need 2 materials (first for shadow caster proxy only) but Unity doesn't have shadow casters culling Front mode available currently.

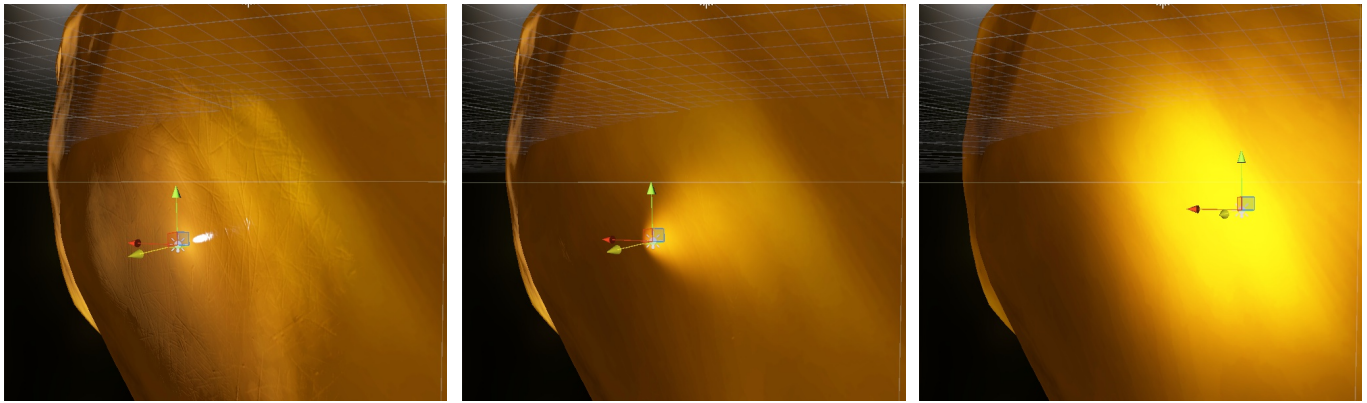
Our setup works like this – actual model receive translucency, because shadow caster proxy is close enough to the light back-facing surface so it's skipped by shadow bias (just like objects don't cast shadows on themselves on light front-facing surfaces). The same moment our proxy casts shadow on other stuff in scene, because translucent objects are not transparent and block certain amount of light passing thru them. However **Suppress shadows** slider described above works reasonably well and probably you won't need such 2 object setup with shadow caster proxy.

NdotL reduction slider is new to UBER 1.03 feature that is esp. useful with thin or 2 sided meshes like vegetation. By default it's 0. When it's 1 and light direction is parallel to the surface normal translucency will be dimmed. Translucency amount is simply multiplied by surface NdotL factor and that can be observed in reality for thin objects like paper sheet where amount of light transported to the light backfacing side depends on surface orientation against light direction.

Spot exponent slider controls the falloff of translucency. Check it for directional light with **Scattering** set to 0. For higher exponent you'll need to point camera exactly against the light direction to see translucency. For lower exponent you will see translucency with wider falloff.

Scattering itself is meant for perturbing light direction taken for translucency with model world normals. With higher scattering your translucency will depend more on mesh topology. For maximum scattering you'll see that only fresnel oriented (steep viewing angle) surfaces receive translucency. So – it's often not needed to provide model with translucency occlusion map. Dial up scattering a bit and you'll break uniform look of transmitted light for more non-uniform and natural look.

Point lights directionality is kind of improvement in UBER comparing to original DICE model, which that paper missed as well (together with shadows treatment):



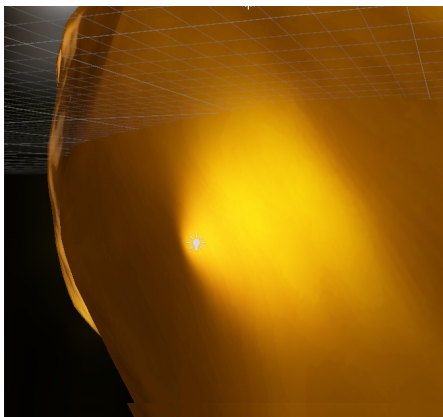
Point light close over surface (incorrect look)

Point light close under surface (incorrect look)

Point light farther under surface (correct look)

With point lights the light direction over the surface is not constant like in case of directional light. When the light is placed close to the surface light direction around changes dramatically. When close and over the surface it still transmit light inside and should scatter there. Instead, angle between viewing and light ray is negative and we get no translucency. Despite of it point light close and over the surface gives narrow falloff as well which is incorrect – as light gets inside the medium, scatters there all around and some amount should get back to eye. Light placed close and under surface gives positive angle and although we get some translucency, its falloff is very narrow no matter how huge **Spot exponent** would be set. The situation gets more correct when point light goes inside the object a bit farther (3rd screenshot above) from surface.

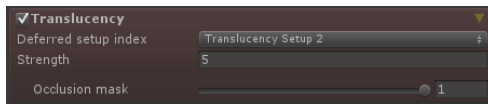
So - **Point lights directionality** slider is meant to gradually change the way light direction is computed. For value of 1 we treat point lights as directional lights and the vector is constant – taken from point light origin to the camera.



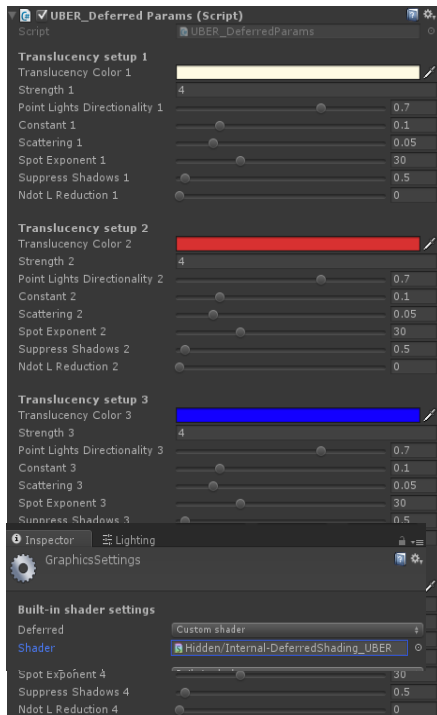
For this screenshot **Point lights directionality** is set to 0.5. Light direction is then something between actual light direction against the surface (light to surface point vector) and constant direction (light to camera vector). Point light is still placed very close to the surface, but we get more natural wider translucency falloff.

Separate discussion should be taken on translucency in deferred lighting (above was right for forward). Deferred G-buffer can't provide such capacity to store all translucency parameters per pixel like exponent falloff, scattering and so on. Actually we barely have one output MRT texture channel available for the purpose. UBER takes mostly unused emission (A) of the G-buffer to store info for translucency (the choice is that it's not commonly used by any 3rd party rendering solution like Alloy so we don't have any interference). G-buffer value we store is only the most important info – translucency intensity and setup index. We've got 4 independent setups available. Without it we couldn't get separate translucency settings for separate objects. In frozen ice like scene it might be white, but think when you'd like to introduce characters with their "red ears" translucency tint. That would be not possible then as we would have to take only one color for every object rendered in deferred (or render characters in forward). UBER comes to help here. We can define up to 4 translucency setups and select which one will be used per object.

That's the **Deferred setup index** dropdown available in UBER material inspector translucency section. It's available when main camera render mode is set to deferred:



You also need to set **UBER_DeferredParams.cs** script on the camera. The script will setup proper command buffer for translucency usage and will set global shader translucency parameters. For example scene it's already set like this:



Translucent objects in the example scene use setup index 1 while ear uses 2nd index (red color). This way ear can have different translucency tint in deferred. As we see parameters that to be set here are the same as in material inspector. The way translucency intensity is written into G-buffer per material is (Color (A) x Occlusion mask x Strength) + setup index. Then deferred lighting function will take the translucency intensity for given pixel, decode setup index and perform regular translucency calculations like it would be in forward mode.

So it's necessary to use both – above script for the camera and dedicated deferred lighting shader provided (take a look at **UBER/Shaders/Deferred_Lighting** subfolder).

If you'd like to use UBER shaders with Alloy's lighting in deferred you need to use the other dedicated shader placed there (info inside and ready made shader that works with Alloy 3.3.1 is included).

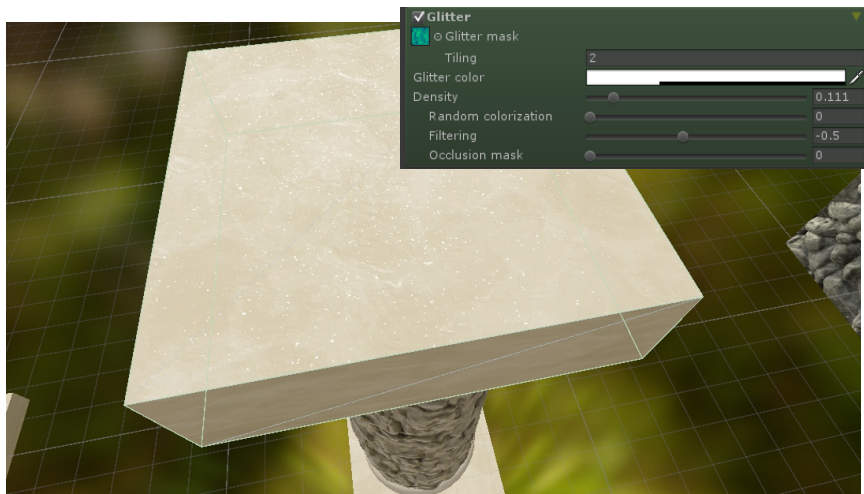


Two steps necessary to use UBER shaders effectively in deferred rendering path are to attach **UBER_DeferredParams.cs** script to the camera and selecting custom deferred lighting shader in Unity/Edit/Project Settings/Graphics. Note that translucency needs to acquire G-buffer (A) channel thru command buffer. If you use multiple cameras they would need the script to be attached as well. Translucency in deferred works currently only for HDR cameras (which are needed for good looking PBR anyway).

Glitter

This feature introduces small glittering sparkles over the surface. Originally it was designed to work with snow only, but I separated it – you can have glittering surfaces without snow used.

On this example we don't have snow and glitter mimics small particles of quartz present in the marble the column top is made of. The effect is based on the glitter mask that's sampled twice with different offset and multiplied. Coords for glitter mask samples depends on object and camera position and orientation. Default glitter texture (R) channel consists of sparsely distributed dark pixels, so only where two bright samples meet together we can see sparkle.

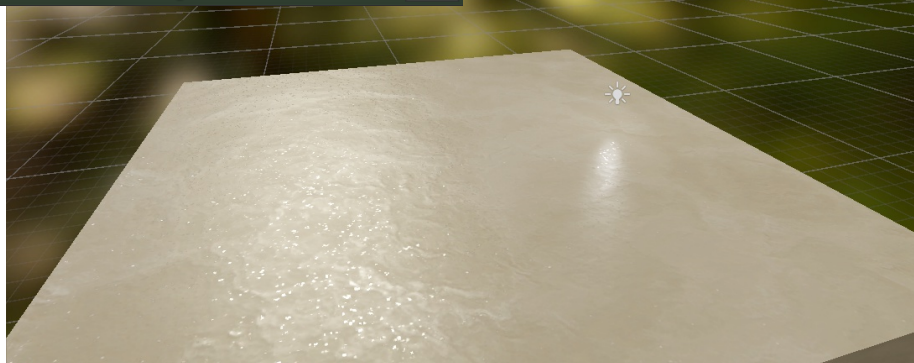
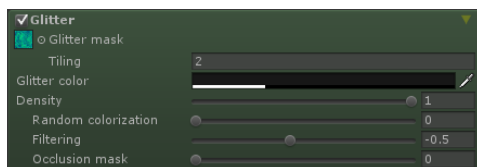


Glitter output is routed to specular color for lighting. Sparkle is then very bright specular pixel and even moderately bright environment reflection probe can produce bright sparkles. Where you see a sparkle you see *first **Glitter mask** sample x second **Glitter mask** sample x **Glitter color** (RGB) x **Density** x "huge constant"* and this value is added to specular output.

Separately we can increase smoothness output for lighting – (A) channel of **Glitter color** is smoothness value added to a sparkle. For completely smooth surfaces it would do nothing then (we can't make it more smooth), but in real situation and objects that are rough (rocks, marble, snow) you specify a bit of smoothness boost from sparkles – this way they match to specular highlights when we look at the surface from the angle the highlight is visible.



Note that Glitter mask texture is shared with snow Micro surface texture. This saves one texture used in shader (important for non DX11 platforms where we can run out of textures available when lightmaps are used).



Glitter color RGB is low, but we see sparkles on specular highlights because of gentle smoothness boost present on that pixels. **Density** has been set to max to emphasize the effect.

Random colorization is useful to gently tint sparkle output depending on world position. It mimics these rainbow gradient seen when incoming white light splits into color components.

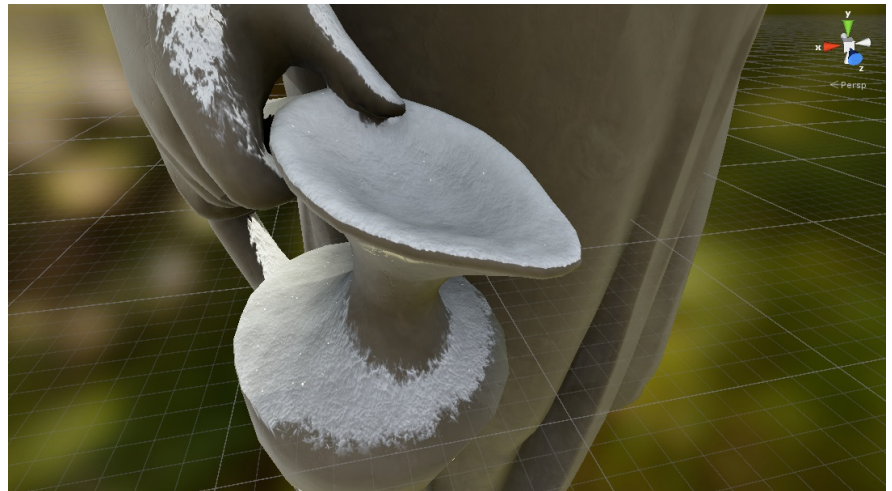
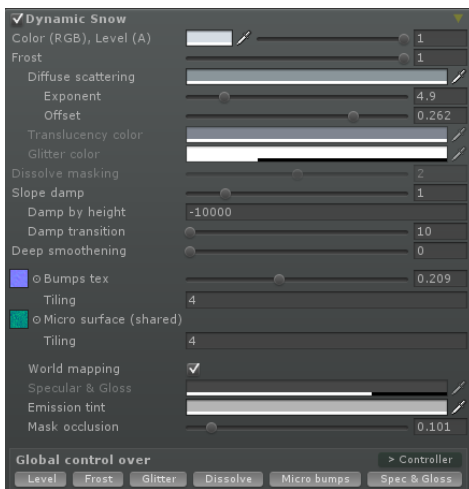
Filtering is MIP offset when sampling **Glitter mask** texture. Set it below 0 to make it more random and noisy when viewed from distance (sparkles can randomly appear even at far distance).

Occlusion mask is amount of masking applied to glitter (parts of the surface might be needed to have no glitter present). This is the same texture channel used as with Translucency feature masking described above, so you can mask translucency and glitter separately then).

When Glitter feature is present you will have an option to select independent glitter color RGB+A for dynamic snow (your surface w/o snow can present small or no glitter while snow will present it more with brighter **Glitter color** values). To have glittering snow you need to select Glitter feature.

Snow

This feature is meant to cover objects with snow – dynamically. For changing weather conditions it is very handy as you don't need to model your snow scene separately. With a single slider you can cover everything with snow gradually.



Untypically dynamic snow is not shader feature handled by shader keyword variants but separate shader. This doubles number of shaders included in the package, but also decreases exploding amount of variants for UBER shader. Without it amount of RAM you need in editor would be doubled as well. When you select dynamic snow feature material inspector will replace shader with the variant that has snow feature present.

Snow has its configurable **Color** (RGB), so actually you could use it for sand coverage as well.

A. Level of the snow is controlled by (A) value of the color. The level slider is also exposed next to the color for convinience (will affect alpha value of the color and vice versa).

B. Next factor for snow level is **Slope damp** slider. Set it to 0 and the snow will can be present on any part of the model regardless of how steep it is. Higher values of **Slope damp** makes snow to be present only on flat parts of the mesh.

C. To have snow present above given “over the sea level” use **Damp by height** and **Damp transition** values. Be default threshold is set to -10000 which means snow will be present on everything that's placed above -10000 world Y position. You can tell the shader to place snow on highlands – for example set **Damp by height** threshold to 3000 and transition to 500. Above 2500 snow will gradually appear with 3000 Y axis world position UBER will stop damping the snow level.

D. Last one factor for snow coverage is vertex color G channel as you might need to exclude mesh parts from being covered by snow (sheltered).



Watch the above 4 coverage factors when you wonder why you actually can't see snow somewhere. Level might be also controlled globally – see below.

Also – as in case of the dynamic water – snow can be also controlled globally by the script mentioned in Water section above. **Global control over** subsection of the material has quick shortcut for selecting global controller script ([>Controller] button) plus 6 parameters that can be controlled in pairs with global values - [Level] [Frost] [Glitter] [Dissolve] [Micro bumps] [Spec & Gloss]. They are paired with subsequent values of **Global Snow** part in the global controller script. In typical scenario for dynamic weather control you will probably need to set them all to be controlled globally.

Global snow level is multiplied by the snow level per material. The same for frost. In case of other 4 switches (Glitter, Dissolve, Micro Bumps, Spec & Gloss) – global values override material values when button switch is selected. That's why that options might be inactive in material.

Frost level controls amount of additional diffuse scattering over the surface. When dynamic water is present the effect will be present on the wet parts only (water that's frozen gives gentle scattering).

Diffuse scattering part is set here independently from the same present in **Main Maps** section. You can set different values for snow diffuse scattering and underlying surface.

Translucency color is active only when **Translucency** feature section is selected. The color of translucency for snow can be managed independently for parts covered by snow. For more info refer to Translucency feature section.

Glitter color is active only when **Glitter** feature section is selected. The color (specular intensity and smoothness gain) for snow can be managed independently for parts covered by snow. For more info refer to Glitter feature section above.

Dissolve masking controls the amount of dissolve effect on micro surface. When it's set to maximum snow will appear coming from separate, hard defined snow flakes (from dissolve mask). When masking is set to 0 appearing snow (low coverage) will look like alpha blended with micro normals applied. For “dry” fresh fallen snow dissolve value is higher as we see separate snowflakes covering surface. For melting snow dissolve value should be lower because separate snowflakes melt and we perceive snow like wet solid medium rather than separate snowflakes.

Deep smoothening causes cancellation of underlying surface normals and bump tex (see below) appears. When snow gets deeper we don't have thin layer of snow anymore but it should rather look like snow covered all underlying bumps.

Bump tex (would be shared with water ripple texture when water is present) holds large scale normals for deep snow. Strength is controlled with the slider on the right side of texture slot. Texture is sampled with selected **Tiling**.

Micro surface (shared with glitter texture) is composite texture that's provided with UBER. You might author it in case you'd like to have different look of snow upclose (for example when you'd like to use dynamic sand instead). (R) texture channel holds glitter mask (that's why it's shared with that feature) as dynamic snow is typically used together with glitter. (G) channel holds dissolve mask – brighter pixels appear first when snow covers surface. (BA) is microsurface normal map that matches dissolve mask. This way single snow flakes that appear on the surface will have correct normals applied. We can select **Tiling** for microsurface texture separately. When **World mapping** is selected snow covers meshes continuously and you don't need to worry about seams for patches of meshes stitched together. This has also advantage that micro / large bumps on the snow have the same size all around regardless of Uvs of the model. The drawback is – it can be used on static models (that don't move) only and we have typical stretching of snow textures on slopes, because mapping is top planar (XZ) in world space. For many applications this is perfectly fine as we remove snow on slopes anyway. When **World mapping** is unselected snow texturing coords are taken from secondary maps section (its UVs and tiling).

Specular & Gloss – (RGB + A) holds PBR values for snow with specular setup. Basically default spec color is fine (dark grey) and moderately high smoothness.

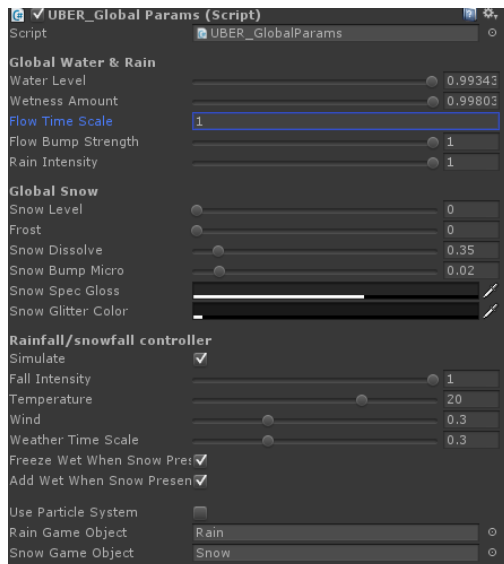
Emission tint is color that emission of underlying material (when used) is multiplied by. With black your snow will not “pass thru” any emissiveness of underlying surface. With white color we'll pass the underlying emissive color as it is.

Mask occlusion can bring a bit of detail – dissolve mask sampled for snow coverage can be used AO value. For high slider values it doesn't look natural, but for low values it can be desirable to bring a bit of occlusion present of micro surface.

As mentioned snow can be controlled globally. Values that needs to be affected for different snow conditions (when it's covering, melting or drying) are selected in **Global control over** subsection.

Global params script controller

This script has been briefly described in Water section above. When certain parameters on the material (for water and snow) are selected to be controlled globally we can handle them here in **Global Water & Rain** and **Global Snow** sections. The dynamic weather conditions are set in **Rainfall/snowfall controller** section at the bottom of the inspector.



When **Simulate** checkbox is selected the script will affect all above water and snow parameters. That's why water level and wetness amount are not set to 1. They are continuously animated – **Fall intensity** for positive **Temperature** means Water Level and Wetness Amount will increase. They would reach 1 but the same time we have drying effect. It's more intensive for higher **Temperature** and **Wind** strength.

For **Temperature** below zero fall will mean global snow coverage increases.

Although the weather controller is not physically based in terms of precise physical values, it works with common sense. For example if a surface is dry and we've got positive temperature – fall intensity set to 1 will give us heavy rain effect. Everything

will get wet (Wetness Amount increasing) then Water Level will also increase (heavy rain means water will start accumulate and flow over objects). Then, let's say temperature goes below zero – water will freeze (**Flow Time Scale** set to 0) and the global **Frost** value will be pushed to 1. If fall intensity will be still positive we'll snow, if not we will see only frozen water. Another example – we've got dry environment and snow start falling (temperature set below 0, fall intensity positive). After we cover the scene with snow fall intensity decreases (snow stop falling). Then temperature goes above 0 and snow start melting. It's dissolve masking, glitter values will be lowered. Glossiness will be increased as melting snow covers with thin layer of the water, micro bumps are also dimmed then. When snow melts completely remaining water will melt and start flowing again. In time without fall (rain) when the temperature is positive global water level will be decreased – scene dry again.

When **Freeze Wet When Snow Present** checkbox is set we won't touch global water level on the scene before global snow level is above 0.05. That's when snow covering the frozen water will “hold” the negative temperature for underlying water.

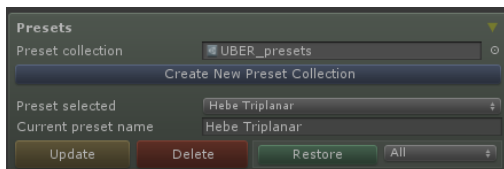
Add Wet When Snow Present checkbox will increase global water level together with snow. Snow will cover the object, but underlying parts will be also covered by wetness. It's useful for drying scenario (snowfall, snow dmelt to water that's introduced by this checkox, water dry when snow melts completely).

You might also notice that Wetness section has checkbox named **Pair level with snow**. When checked you can set wetness to be present wherever snow is present on the material. Useful for surfaces that you want to remain wet after global snow level is down (end of global snow melt cycle). You need to enable snow and global control of snow level to make **Pair level with snow** switch available. You'll have wetness present only on parts where snow is present on per material basis. Although snow might be not visible, because global controller set snow level to 0, wetness would be still present where snow would be if global snow level is 1. This might seem to be complex, but actually you can think about such cycle – dry object, snow starts falling and covers only top parts of the object. Together with snow appearing on the surface we increase level of the water. It gets covered by snow so we actually don't see it's wet under the snow. Then snow stops falling and we reduce global snow level to zero (snow melts with temperature). Parts where snow was preset remain wet (we set wetness to 1). Other parts of the object are still dry because we paired water level with snow which covered object partially (for example snow covered slopes only). Now we reduce gradually wetness to zero – remainings of melted snow dry and the snow cycle is over. And this can be achieved by global script weather controller with only fall intensity and temperature values ! Set environment dry temperature below zero. Increase fall intensity (snow falling). Stop fall and increase temperature back – snow will melt and remaining water will dry on the parts previously covered by snow.

Use Particle System checkbox makes it possible to control two particle system prefabs that are included in UBER package. They are present on the example scene as **Rain** and **Snow** game objects. When you set the script references to these game objects it will automatically control rain and snow fall intensity depending on temperature and **Fall intensity** simulation value. Observe how it makes active/inactive and control particle system emission intensity when you're in play mode on the example scene.

Presets

Presets are very handy when you'd like to test some material parameters and be able to save current state of shader w/o need to copy material. It can also copy parameters for distinct feature like wetness, snow – this way all your materials can have the same properties and you don't need to copy them

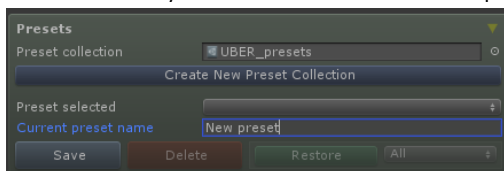


manually parameter by parameter across scene.

Presets are stored in asset file called **Preset collection**. By default UBER comes with default empty collection placed in UBER installation folder (**UBER_presets.asset** file). You can work on this collection or make your own using **[Create new Preset**

Collection] button. **[Update]** button is active when you've got a preset stored and then you can update the preset.

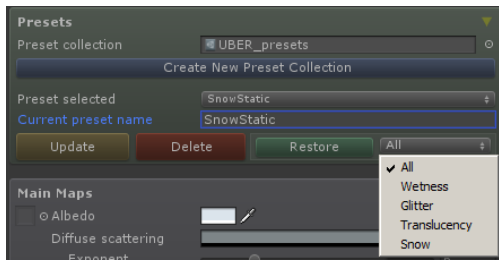
When **Current preset name** entered can't be matched with a preset in collection (or collection is empty) it's assumed you want to create new preset:



[Update] button is now titled **[Save]**. So – names of presets in a collection are unique. **[Delete]** button is active when an existing preset is selected. The same for **[Restore]** button which will copy back parameters from preset to current material.

[Update], **[Delete]** and **[Restore]** buttons will always prompt when clicked unless you hold SHIFT when pressing them.

Right side dropdown has following items:



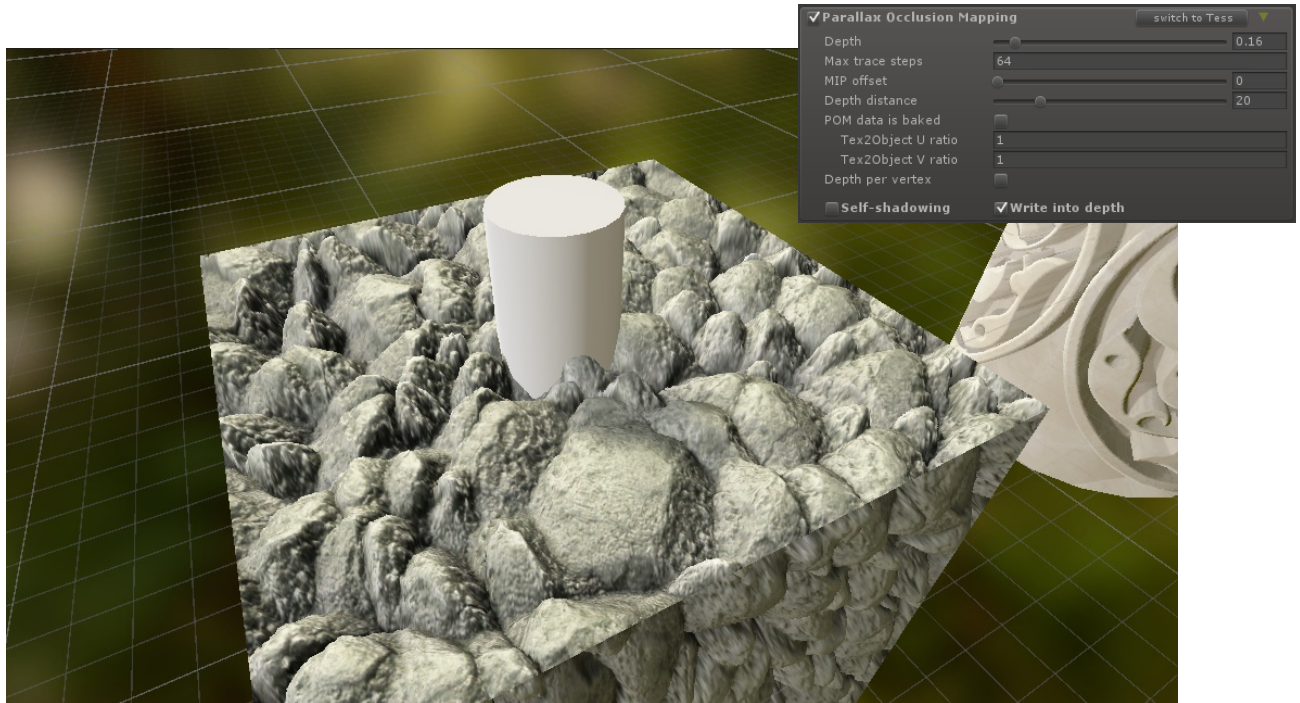
All – restore all material parameters and shader reference
Wetness – restore material parameters that belongs to water
Glitter – restore material parameters that belongs to glitter
Translucency – restore material parameters for translucency
Snow – restore material parameters for dynamic snow

You can then copy whole material state (All) or for selected feature. Let's say you tweaked water parameters so you're satisfied. In the water “template” material save preset, then select **Wetness** from dropdown on the other object you'd like to copy water parameters to and click **[Restore]**.

Notice that with a preset collection you can simply share presets between objects that are placed within different scenes (like it would be “collection of materials in project” but much more handy to use).

2. POM & Tessellation

UBER comes with variety of advanced parallax shaders, this chapter covers ideas and Parallax Occlusion Mapping shader types available. As POM is generally Core functionality (it differs for tessellation shaders, POM Advanced and Extrusion variants), let's start with POM section in Core UBER shader.



When Parallax Occlusion Mapping is disabled you can use default Parallax Mapping with displacement slider placed at the right side of Height Map in primary maps section. Default Parallax mapping differs from Standard Shader implementation with the direction – it's always made inward (white pixels in heightmap are not displaced). With POM feature enabled you've got options like on the inspector screenshot above. It belongs to the material of example cube shown. Note that the section is not visible when your object has no heightmap texture assigned.

Depth is obvious – it's displacement strength (displacement is realised inward).

Max trace steps controls quality vs. performance. Notice that the steps are maximum limit for tracing iterations inside shader. Larger number set doesn't mean all steps will be performed. Shader stops searching when it hits displaced surface.



For DX9 and OpenGL platforms number of maximum steps taken are fixed to 256 and **Max trace steps** slider does nothing. This is because of bugs in Unity that miscompiles loops for DX9 and can cause openGL machines crash when number of iterations is variable. You'd like to check if it actually works (with newer Unity release it might be fixed). Look for `SAFE_LOOPS` define in `UBER_StandardConfig.cginc` file and comment this define out.

Displacement values from heightmap are taken from subsequent MIP levels depending on distance and viewing angle, so for distant objects don't worry about POM performance. In most cases it will reduce to 1 step search. This is because heightmaps have different MIP levels imported. They are not average, but maximum value of 2x2 pixel matrix from the MIP level below. That's why your heightmaps should have " _Heights" sequence in their name. This way UBER texture importer will process higher MIP levels for heightmaps automatically. With **MIP offset** slider you can increase heightmap MIP level

that POM tracer taken values from. This can increase performance (larger trace steps), but resolution of your high frequency details of surface might be degraded. Adjust it to taste then trying to maximize **MIP offset** value. Additionally you can reduce POM effect at distance to save performance using **Depth distance** value. This is the distance at which displacement depth will be gradually reduced to zero.

One of the most important factors for correct POM mapping in UBER is relation between tangent/texture space and world space. This is because we would like to know how long is the ray solved from mesh surface (where POM tracing starts) to the point it hits “a bump” in the heightmap texture (where POM tracing ends). It is known in texture space (UV parallax offset), but we'd like to know what's the distance in real world space. This is necessary to correctly intersect POM objects with the other when **Write into depth** checkbox is selected. If texture to object UV ratios are incorrect the intersection will not work. For some simple objects like boxes, cylinders we can specify it manually like on the example box above (**POM data is baked** checkbox disabled). The default Unity's Cube mesh has extents of 1 unit in object space. The same time it's mapped so that cube walls are fully mapped into 0..1, 0..1 UV coords. This means that 1 unit distance in texture space maps to 1 unit in object space. That's why **Texture2Object U ratio** and **Texture2Object V ratio** are set to 1 and intersection between cube and cylinder in the center works correct. For another primitive like plane ratios are 10,10 because this mesh has extents 10x10 in object space while it's mapped to UVs 0..1,0..1. Simply add a plane and put UBER POM with z-write material on it. You see how **Texture2Object U/V ratios** works. For values other than 10,10 intersection won't work correctly. You might ask – if this mesh property (size and mapping) is referenced to material it will be not possible to use the same material with different meshes. That's right – use it on simple objects where you use separate material for. If you'd like to use one material for multiple objects (for sake of batching) you need to bake ratios into mesh vertex data (UV4 set is used for this). This happens automatically when model is imported – the only thing you need to do is name mesh model file with **_POM_Baked** suffix. UBER mesh importer will process it for you automatically filling UV4 with correct data. Refer to example **torus_POM_Baked** asset. If you'd change the name so it doesn't have **POM_Baked** suffix it will break the shader behavior. For POM baked meshes you select **POM data is baked** checkbox in material inspector.

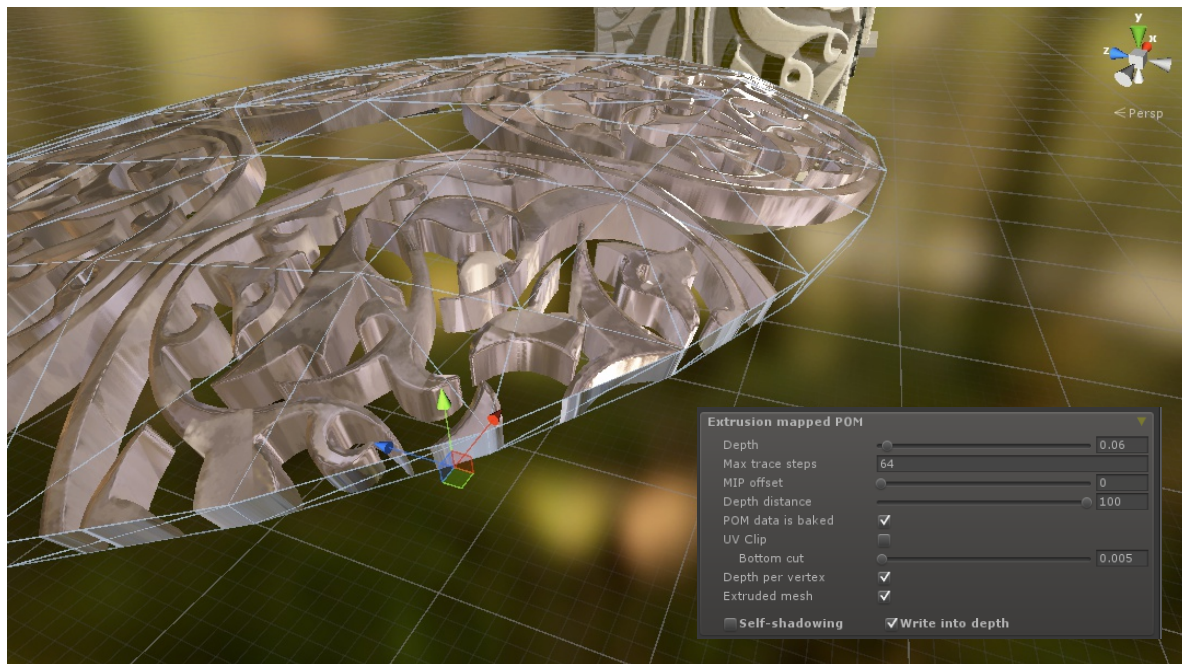


Follow naming convention of models you'd like to use with parallax techniques. Use **_POM_Baked** suffix for UBER to prepare it for POM rendering (note - it uses UV4 in mesh buffer).

Depth per vertex you'll probably won't use unless you'd like to achieve some manual optimization over POM on the surface. Look at the picture on the left. We see mesh surface from the side (bold lines). The arrow is viewing direction – ray traced in POM algorithm. When you know that part of the surface height texture stays always below some level (bottom bold line) you can set this level side vertices. POM ray-tracer will skip the empty space (green part of the ray arrow) within calculations causing performance increase. This is general rule in parallax occlusion mapping that the cost of the algorithm is the cost of travelling thru “empty” texture space where ray stays above the virtual surface of heightmap. UBER doesn't provide tools for managing such optimization and would need to handle vertex (A) color for starting level yourself. Top level is 1 (white heightmap pixels), bottom level (black heightmap pixels) is 0.

Except for POM optional optimization described **Depth per vertex** is used for UBER shaders with “**POM Extrusion Map**” variant (separate shaders in package). (A) channel of vertex color is used there to instruct shader that some parts constitute bottom of extruded mesh (0 in (A) vertex channel). This is managed automatically during import when you specify your model name with “**_POM_Extrude**” suffix. There are a few models included in the package that uses this – for example **Plane_Collider_POM_Baked_POM_Extrude_1_BOTTOM**. Additional **BOTTOM** suffix instructs UBER to cap the bottom of extruded mesh and value (1) is used optionally for colliders only. Bottom mesh vertices will be extruded (displaced vertices in mesh), while without it flat extruded mesh will still stay flat when rendered with regular shader. UBER can modify such doubled flat vertex positions so they

match displacement value of parallax effect. An example are extruded intersecting disks. They have extruded mesh (with `_POM_Extrude` suffix) – on the preview it's flat, but actual number of vertices is doubled and UBER can displace bottom ones inside shader vertex function.



In case of shader variant for POM extrusion map (screenshot above) **Depth per vertex** is always coupled with **Extruded mesh** checkbox. It has additional **Bottom cut** parameter which allows to remove the bottom of extruded mesh (it can have holes).

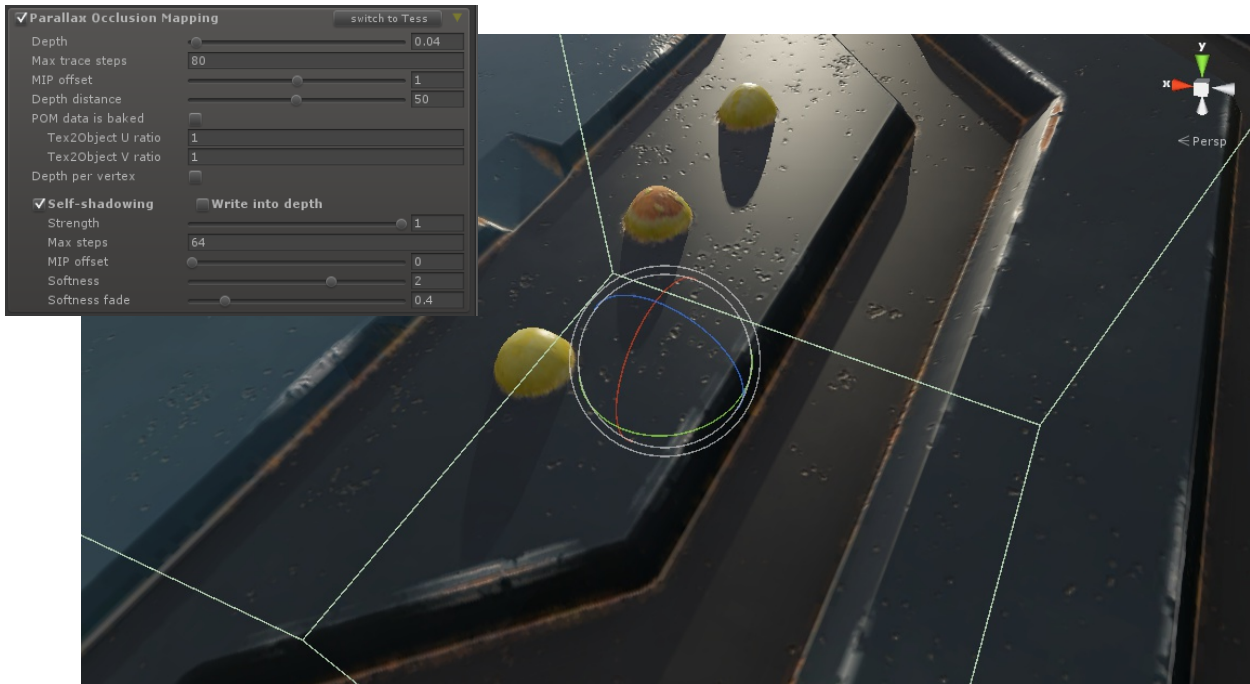
While **Write into depth** option is very cool, you might consider not using it. The reasons are both – performance and lack of resolution for shadows. Writing into z-buffer breaks GPU optimizations when objects are rendered front to back. With z-write this optimization doesn't matter (yes, I've tried to use conservative depth write for DX11 and although it can be compiled it didn't work for me – output was buggy). Additional note is that in forward rendering path POM is always rendered twice – 1st time as depth buffer that's used for shadow collecting and 2nd time for regular camera view. Additionally POM calculations are performed for every light in the scene which cast shadows (we need to modify shadow map for POM object that casts shadows).



Performance wise for POM z-write shaders is using deferred rendering and such POM objects should not cast shadows where it's not visually critical. Consider using proxy objects for casting shadows only instead.

POM z-write works from the perspective of every light that's why such POM objects can be self-shadowed.

2nd reason you'd like to use **Self-shadowing** checkbox instead of **Write into depth** is better self-shadowing shadows resolution.

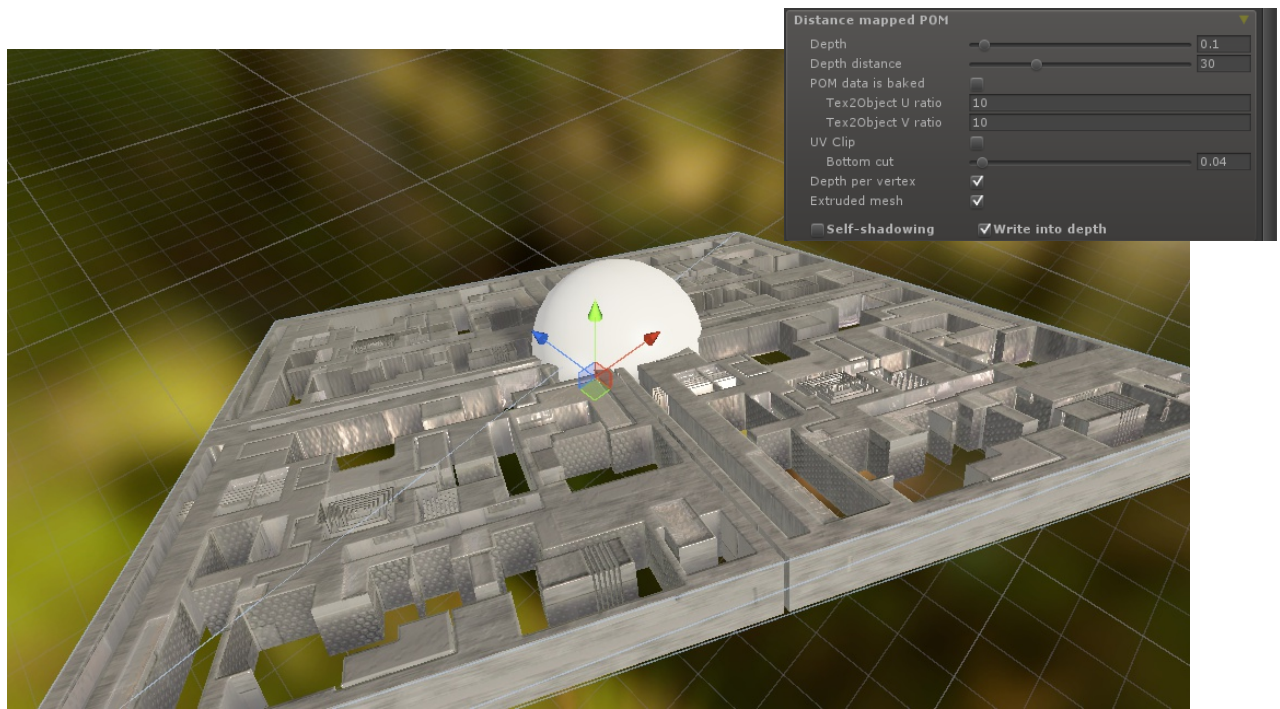


Self-shadowing doesn't rely on Unity shadow maps and their limited resolution, but ray traced self-shadows are rendered instead. At the above screenshot for the example object we see similar parameters like on POM section above with **Strength** that controls level of light occlusion for self-shadows. **Softness** and **Softness fade** controls self-shadow softening. Notice that resolution of small texture bumps have very detailed shadows. It could not be achieved with classical shadow mapping.

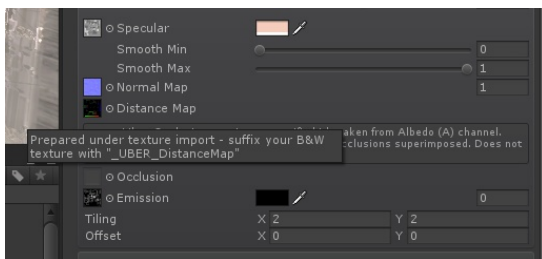


You need to use dedicated lighting shader in deferred to get self-shadowing working. Shader is placed in UBER/Shaders/Deferred_Lighting subfolder. Place it in Unity/Edit/Project Settings/Graphics – custom deferred lighting shader slot (was already discussed with translucency for deferred in previous chapter).

Below screenshot refers to **"POM Distance map"** shader variants.

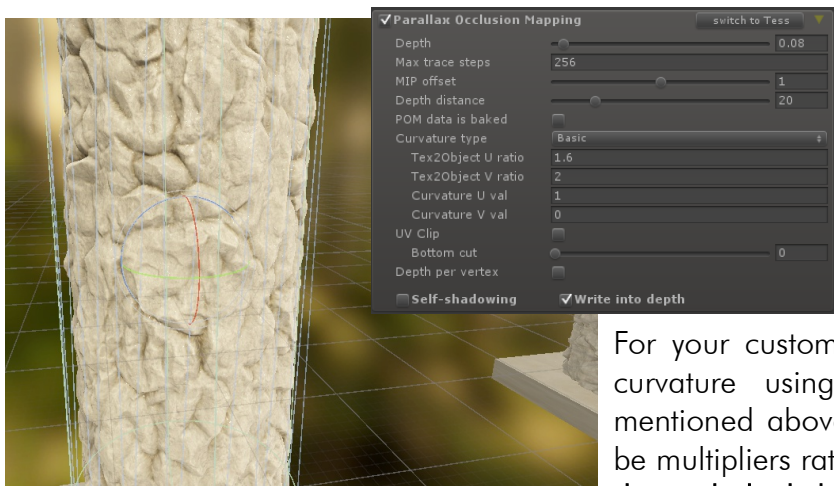


Control over POM section is the same as in case of Extrusion map shaders, but the extrusion is done specific way – it's always parallel to either U or V axis in texture space. Although it is obvious limitation, but for some class of objects like above sci-fi example (or “space station” example) it looks fine. The advantage is very good performance. POM tracing can be solved in 2-3 steps for typical situations. That's why this POM technique is very fast. Base for calculations is not grayscale heightmap like in the all other POM shaders, but distance map instead:



Distance map is automatically prepared at texture import level when filename follows suffix rule - “_UBER_DistanceMap”. When you'd look at **spaceShip_borders_UBER_DistanceMap** file outside Unity you'd see B&W 256x256 texture with black pixels that outlines extruded parts of the surface. UBER texture importer gets B&W 256x256 texture and turns it into distance map that's used by shader.

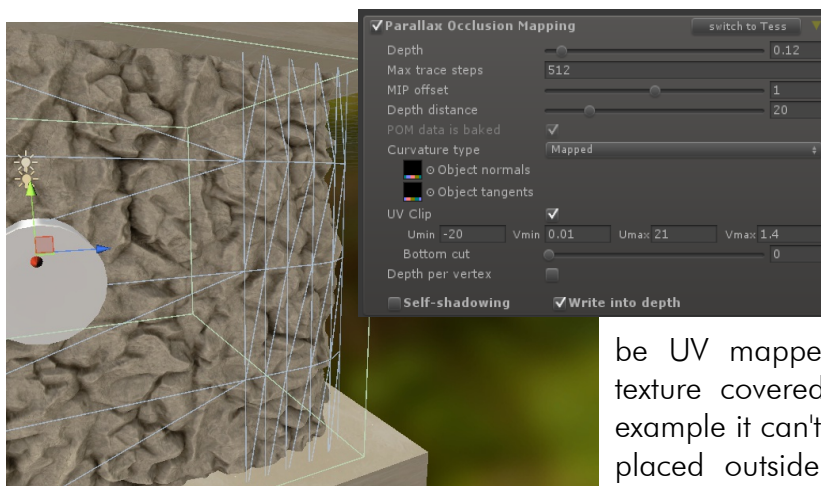
Another shader for POM is **POM Advanced** variant. It can work in 2 modes – simple (best for objects of constant curvature over the surface like cylinders) and it's quite a cheap technique to solve silhouettes:



Curvature type is set Basic. Texture to Object ratios are set manually and follows cylinder size/mapping (for z-write intersections to work). Look at **Curvature U val**. It's set to 1 and defines curvature of the object along U texture coordinate. As this is cylinder V curvature is set to 0.

For your custom objects you could bake ratios and curvature using POM_Baked mesh model suffix mentioned above. Then Curvature U/V values would be multipliers rather than absolute values (select **POM data is baked checkbox** for baked meshes).

What works fine for cylinders could not work fine for more complex objects (like torus) which have curvature values variable over the surface. For torus it's like bursted out curvature (positive) on the outer and saddle like curvature (negative) in the center – U curvature value changes from positive to negative while V (along the circle) is constant. Change torus example curvature type to Basic from mapped and you'll see. To solve this problem UBER comes with solution – mapped curvature type used on torus and “corners solved” examples. It requires mesh normals and tangents to be baked in texture.



To get textures required right click on the mesh filter component header. **Bake Object Topology to Texture** option is available there.

For POM Advanced shader it will also fill the texture slots in material. Although for torus object mapped POM silhouette is not problem any because it can

be UV mapped continuously without seams (whole texture covered by encoded vectors), for the other example it can't be. We see empty black parts that are placed outside UV bounds. The ray when tracing

silhouette running in POM calculations can go outside the borders at the top and bottom parts of the model, that's why I'm using **UV Clip** bounds to cutout pixels there. This helps removing pixels of the surface that are placed outside certain UV coordinates (after tiling applied).

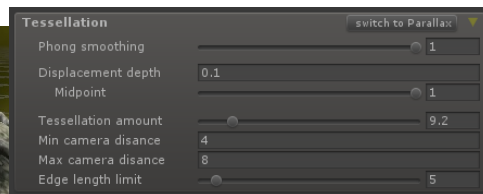
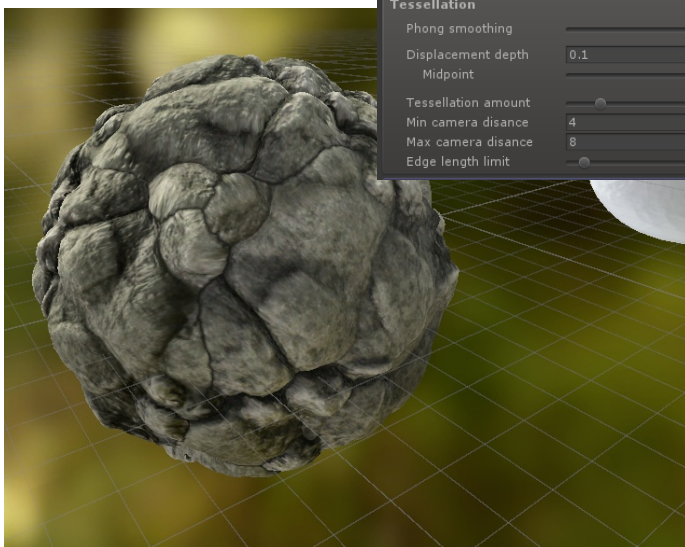
Notice that POM Advanced mapping works fine for certain types of objects (in this example torus), it's the most expensive POM technique available. As normal/tangent maps are material parameters you can use it on one object only unless you map your models to work with atlased UVs. Values of vectors for normals/tangents stored in textures are sensitive to going "out of bounds" like in the example above. So – it might be better to use tessellation to solve silhouettes where it is possible (no need to manage additional textures).



Most of UBER shaders have **[switch to Tess]** button available at the header of POM section. It's available when tessellation variant for shader is available. Instead of selecting the tessellation shader variant manually you can do it simply by clicking the button. The same is for Tessellation – we can go back to POM variant using **[switch to Parallax]** where available.

Tessellation

On DX11 (PC, XBOX One, PS4) or OpenGL4 (Macs) tessellation should be available. UBER allows for using it for 2 purposes – displacement and phong smoothing:



On this example we present tessellation with triplanar selective variant described later.

Phong smoothing allows to smooth low poly mesh after tessellation step.

Displacement depth is obvious while **Midpoint** controls whether displacement will be realised fully outward (midpoint=0), completely inward (midpoint=1) or in-between. For midpoint=0.5 vertices that have heightmap values >0.5 will be displaced outward (in the direction of mesh normal), for heightmap values <0.5 will be displaced inward (opposite to the mesh normal).

Tessellation amount is number of tessellation subdivisions to be done. **Min/max camera distance** allow to reduce tessellation at distance (to save performance because it makes to sense to displace heightmap detail far from the camera – we wouldn't see it anyway). Additional controller is **Edge length limit** – subdivided triangle edge length won't go below such distance (in screenspace pixels). It's useful because your model can consist of small and large triangles. Small polys would go into sub pixel size for large tessellation amount values (waste of GPU power) while big triangles might be not enough tessellated. Edge limit is then kind of "safety performance saver" when millions of too small triangles might appear pushing your GPU to knees (and dropping your performance to seconds per frame instead of frames per second).



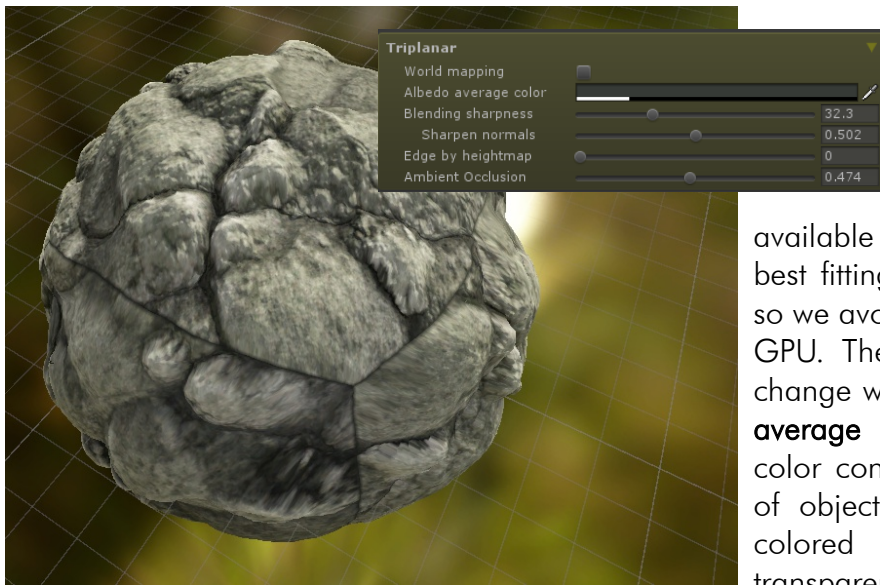
You can configure tessellation shader variant to control amount of tessellation and displacement strength via selected vertex color channels. Look at the header of shader between CGINCLUDE and ENDCG block for more info.

3. Triplanar, 2 Layers & additional stuff

In this chapter we'll cover some other UBER features like triplanar mapping, shader variant to paint 2 independent texture layers thru vertex color and additional goodies.

Triplanar selective

Triplanar selective is new technique for triplanar mapping where instead of sampling texture from all 3 planar directions we do it only once but select the best fitting one. Triplanar mapping is based on the normal direction of mesh to avoid typical texture stretching with planar mapping.



In the example scene we've got the sphere object tessellated that uses triplanar selective. Triplanar is available w/o tessellation as well, but it has parallax mapping available then only. As mentioned – only the best fitting planar coordinates are selected, so we avoid increasing the costly shading for GPU. The edge where mapping directions change we see with dark grey color (**Albedo average color**). Alpha component of the color controls edge opacity. For some kind of objects you could like to not use the colored edge and make it completely transparent. However – for most organic

stuff like rocks here it works quite fine. If you select the color that matches the average diffuse color you can hide the edge effectively. In our case the edge acts as the edge for adjacent rocks. I intentionally set the **Edge by heightmap** value to zero to show you the idea behind the technique. In real situation we don't like straight edge and that's this parameter for. It controls how much detail heightmap modulates the edge – you can think about it as kind of heightblending.



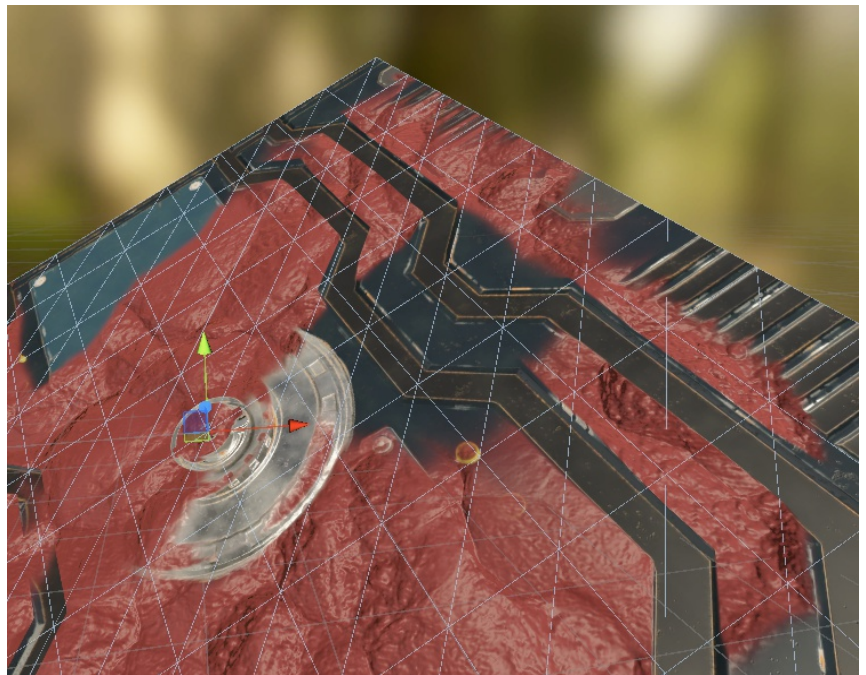
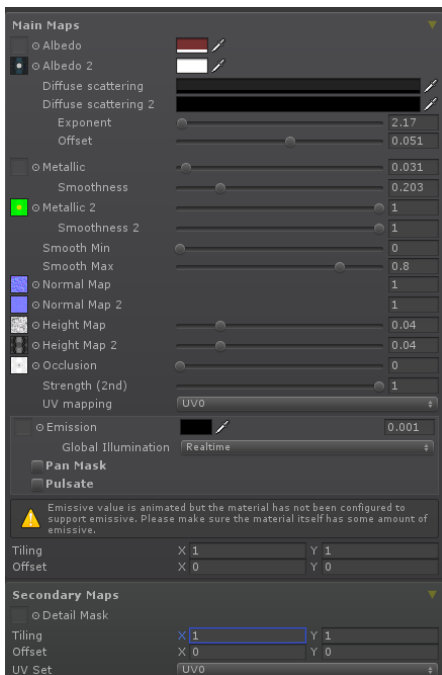
With edge modulated we can very nicely hide the place where two mapping directions meet together. Especially for such kind of texture where the edge acts like edge extruded between rocks and has something like AO placed there (baked into diffuse). **Blending sharpness** controls the width of the edge with additional **Sharpen normals** parameter which can reduce the edge width for normals sampling alone. **Ambient Occlusion** is how much AO indirect lighting will be placed on the edge.

World mapping – use it for static objects as it works faster than in local object space.

Our rocky sphere can rotate in the example scene, that's why **World mapping** is switched off. So the conclusion is that Triplanar selective needs only 2 additional heightmap texture sampling (to modulate the edge) and can be used on any mesh that has normals. No need for seamless UV unwrapping in modelling software – you'll find making rocks and cliffs very easy in your project now.

2 Layers

This is the UBER shader variant that allows you to have 2 independent set of textures and alternate them thru vertex color channel (R). It's not like secondary map where you don't have separate heightmap. 2 Layers shaders doesn't have detail (secondary) maps because shader might run out of textures available (while up to 7 textures are used internally by Unity for lighting when all types of GI baked maps are used, so we've got only 9 textures to be used in shader in pesymistic case).

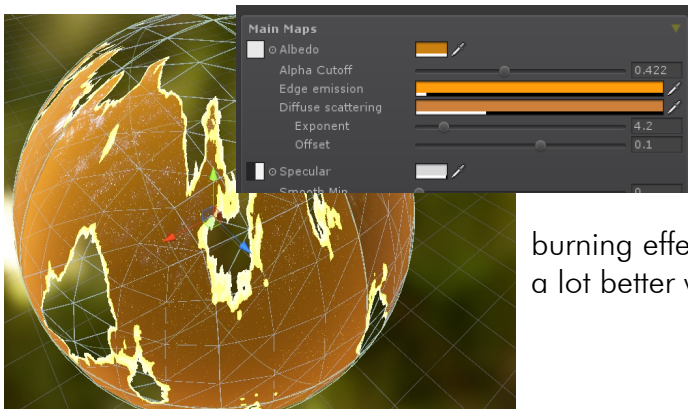


As we see **Secondary Maps** section is reduced to control the 2nd set of textures (2nd layer) tiling and UV mapping coords only. Instead **Main Maps** are larger. In example scene I've used metallic setup for the 2-sided plane object. We've got separate set for most of textures, but in case of **Height Map** we can also use combined texture – (A) channel for 1st layer and (G) channel for 2nd layer when **Height Map 2** is empty. Occlusion texture is always considered to be combined one (1st layer (G) channel, 2nd layer – (A) channel). For additional masking (translucency, glitter) (R) for 1st layer and (B) for 2nd layer are used. We don't have separate textures for **Emission**.

On the screenshot above we can see that blending between layers are not only vertex color dependent but also it's heightblended (basing on layers heightmaps – “higher”/ brighter values of heightmap wins over the other layer). It helps reducing the missing blending resolution across large triangles.

Edge emission with cutoff blending mode

When you select the topmost dropdown of material inspector – Rendering Mode to Cutout, additional parameter (except for alpha cutoff threshold) appears.



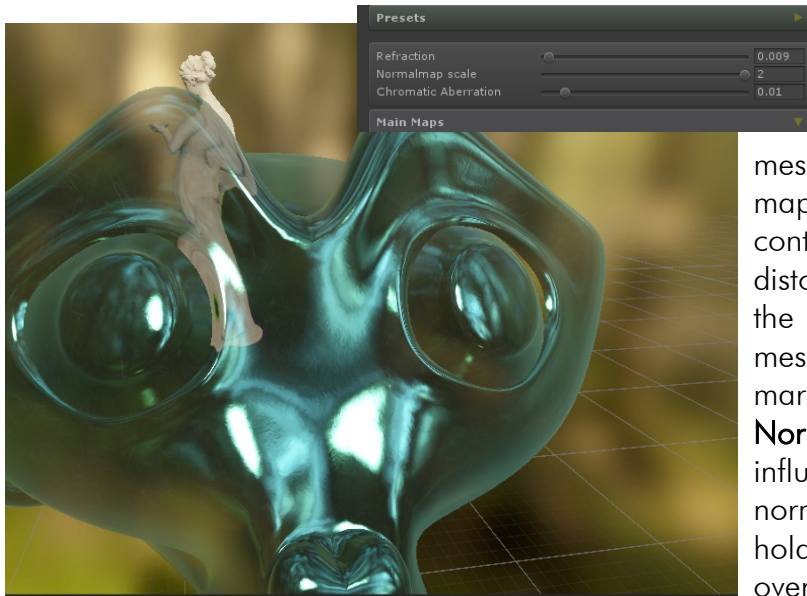
For the example “Carpaint” object **Edge emission** parameter is available. It has HDR RGB value (multiplied by configurable constant) and Alpha color component which controls the size of the emissive edge. The feature might be useful when you'd like to achieve dissolve burning effects like in UBER example scene (of course it looks a lot better with HDR bloom postFX).

2 Sided variants

New in UBER 1.03 you'll see 2 Sided shader variants subfolders. They are managed for core, 2 layers and triplanar variants together with optional tessellation. Useful for vegetation that use translucency as well (an idea might be masking it for leaf nerves used with low translucency exponent). Introduced **NdotL reduction** parameter in translucency is good choice for thin 2 sided material.

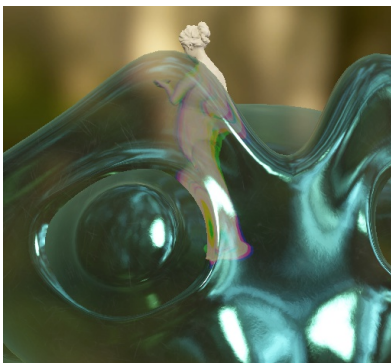
Refraction shader variants

Refraction in UBER relies on grabpass texture, that's why it needs to be selected as separate shader rather than a feature keyword variant. When we've got such refractive shader selected separate section to control it is available. Let's look at Blender's Suzanne example included:



Refraction slider controls amount of refractive distortion. It's multiplier for refraction that comes from both – mesh normals and detail normals (Normal map). With **Normalmap scale** you can control how much the direction for refractive distortion is taken from normal map. With the value of 0 all distortion comes from mesh normals. Suzanne model has gentle marble noise bumps applied. I set **Normalmap scale** to maximum so they influence the refractive distortion (with normalmaps that are very strong you can hold the Normalmap scale low to not overuse it).

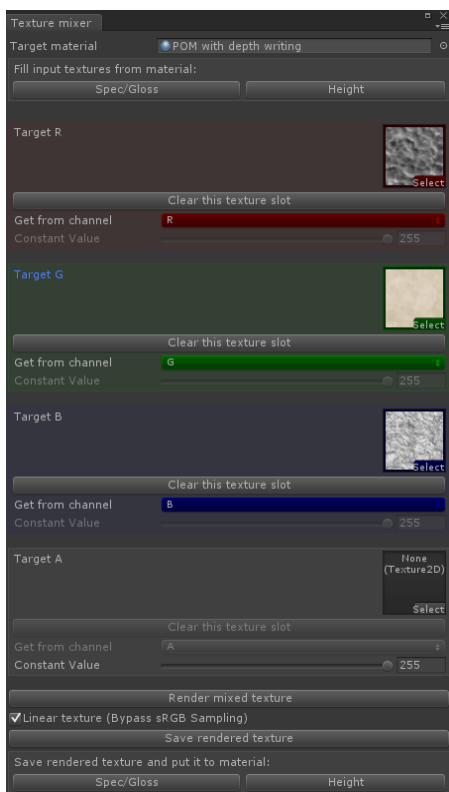
With **Chromatic Aberration** set to positive value we enable the feature which can split RGB color components from grab texture sampled:



Notice that all shader variants have refraction available. For example tessellation with 2 layers and triplanar selective, with snow and water enabled together with refraction is available for both specular and metallic setups. You might make nice refractive glass block covered by snow and with water flowing on its surface. Not to mention refractive materials with water flowing can act well for windows and shields.

Texture channel mixer

You can access this tool using **[Open texture channel mixer]** button placed at the top of UBER material inspector.



Often texture input for shader has to be combined. For example Ambient Occlusion and Translucency mask, Heightmaps for 2-Layers shader variant or metalness/gloss map or even setting transparency data in (A) channel of diffuse texture. You can combine them in imagin software like Photoshop, but it's not much convinient way for things that you ned to manage quickly and in huge amount for larger projects. Texture channel mixer helps.

Output texture is RGBA (although empty A channel you can compress to DXT1 w/o 4th channel data). This is depicted as colored boxes sections named **Target R** to **Target A** on the left. Each one has texture slot for source. When source texture is selected you can select which channel is chosen as input for target channel.

For example if you select a texture in **Target R** section and choose **Get from channel** dropdown (G) then target texture mixed (R) channel will be taken from source (G) channel texture.

If a texture slot for target channel is empty you can specify constant value output. In the example on the left – **Target A** section source texture slot is empty and value writen into rendered/mixed texture will be 1 (255).

When it's possible (all needed data selected and source textures of matching sizes) **[Render mixed texture]** button becomes active. After rendering you can **[Save rendered texture]** into file.

As source textures are often the textures that are currently used in a material where you clicked the **[Open texture channel mixer]** button (it's shown in **Target material** field), ou can automatically fill source textures with a selected texture type from material (buttons available when textures are not empty in material). They might be Spec/Gloss (Metalness/Gloss) texture, Heightmap, Occlusion. Buttons are available at the top of the window tool. After rendering instead of just saving the resultant texture “somewhere” in the project and use it later, you can save it and replace automatically current texture type in material. For example you can tweak smoothness map in an authoring software, save it and re-render in the channel mixer tool. After using **[Spec/Gloss]** button at the bottom of the window new combined texture will be placed in Specular texture slot back in material Primary maps section.

4. Customization

UBER comes with kind of most default state when you import it. However there are a lot of features and selectors that can be tweaked inside shader code header (look for first occurrence of **CGINCLUDE** keyword right after properties) or globally in **UBER_StandardConfig.cginc** file. Depending on shader type you'll be able to tweak shader behavior. For example in Core shader the defines section looks like this:

```
// you can override global blend mode here (per shader - it's globally set in
// UBER_StandardConfig.cginc)
// available modes - _DETAIL_MULX2 _DETAIL_MUL _DETAIL_ADD _DETAIL_LERP
// default mode - _DETAIL_LERP
// #define _DETAIL_MULX 1

// switches used in DEFERRED only
// self-shadows from ONE realtime light (choose the light and attach
UBER_applyLightForDeferred.cs script to it)
#define _POM_REALTIME_SELF_SHADOWS
// self-shadows from Directional Specular lightmap
#define _POM_BAKED_SELF_SHADOWS

// uncomment if you want to save one sampler on platforms with limited texture (Dx9/OpenGL)
// this will disable cubemap blending, be we gain one additional texture sampler for a feature
// #define UNITY_SPECCUBE_BLENDING 0

// if defined you can control translucency by vertex color channel
// #define TRANSLUCENCY_VERTEX_COLOR_CHANNEL a

// if defined we can control occlusion by vertex color
// #define OCCLUSION_VERTEX_COLOR_CHANNEL a

// available r, g, b, a, comment out if you don't want to control it via vertex color
#define VERTEX_COLOR_CHANNEL_DETAIL r

// 1st layer coverage in _TWO_LAYERS mode
#define VERTEX_COLOR_CHANNEL_LAYER r

// wet coverage
#define VERTEX_COLOR_CHANNEL_WETNESS a

// ripples strength (when enabled in material inspector and WET_FLOW above IS NOT defined)
// #define VERTEX_COLOR_CHANNEL_WETNESS_RIPPLES b

// flow strength (when enabled in material inspector and WET_FLOW above IS defined)
// #define VERTEX_COLOR_CHANNEL_WETNESS_FLOW b

// wet droplets strength (when enabled in material)
#define VERTEX_COLOR_CHANNEL_WETNESS_DROPLETS b

// snow coverage (when enabled in material)
#define VERTEX_COLOR_CHANNEL_SNOW g

// glitter mask - used from vertex only if defined below
// #define VERTEX_COLOR_CHANNEL_GLITTER g

// you can use below switch as diffuse color (albedo) tint
// #define VERTEX_COLOR_RGB_TO_ALBEDO
```

When you look at comments placed at the top of every **#define** keyword switch you quickly see that many things are adjustable. Changing it here will change the way it works for all objects that uses the shader. If you'd like to limit the effect for a group of objects – copy the shader, you can place it in dedicated **__UserCustomizedShaders** subfolder placed in UBER/Shaders. Then I'd suggest to change the name of shader inside so you can recognize the copy when you select the modified shader. Watch the dependencies as well. For example for dynamic snow to be available you need to have the twin copy of the shader with “- DynSnow -” shader name descriptor. Otherwise you won't be able to turn on snow feature or **[switch to Tess]** button might be inactive.

Other than per shader you can control UBER behavior (emission multiplier constants, various switches, etc.) globally tweaking **UBER_StandardConfig.cginc** file. Look into it. All switches are commented the way you shouldn't find it difficult to figure out what they do. For more specialised in-depth info you can ask me on the forum thread.

