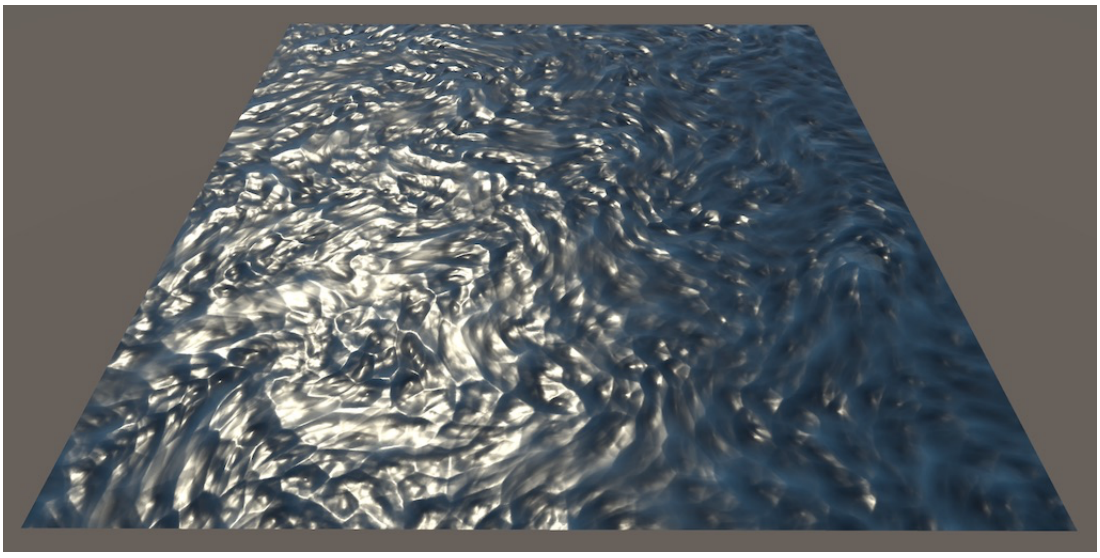
Texture Distortion Faking Liquid

Adjust UV coordinates with a flow map.
Create a seamless animation loop.
Control the flow appearance.
Use a derivative map to add bumps.

This is the first tutorial in a series about creating the appearance of flowing materials. In this case, it's done by using a flow map to distort a texture. This tutorial assumes you've gone through the Basics series, plus the Rendering series up to at least part 6, Bumpiness.

This tutorial is made with Unity 2017.4.4f1.



Stretching and squashing a texture in multiple directions.

1 Animating UV

When a liquid doesn't move, it is visually indistinguishable from a solid. Are you looking at water, jelly, or glass? Is that still pool frozen or not? To be sure, disturb it and observe whether it deforms, and if so how. Merely creating a material that looks like moving water isn't enough, it actually has to move. Otherwise it's like a glass sculpture of water, or water frozen in time. That's good enough for a picture, but not for a movie or a game.

Most of the time, we just want a surface to be made out of water, or mud, or lava, or some magical effect that visually behaves like a liquid. It doesn't need to be interactive, just appear believable when casually observed. So we don't need to come up with a complex water physics simulation. All we need is some movement added to a regular material. This can be done by animating the UV coordinates used for texturing.

The technique used in this tutorial was first publicly described in detail by Alex Vlachos from Valve, in the SIGGRAPH2010 presentation *Water Flow in Portal 2*.

1.1 Sliding Surface Shader

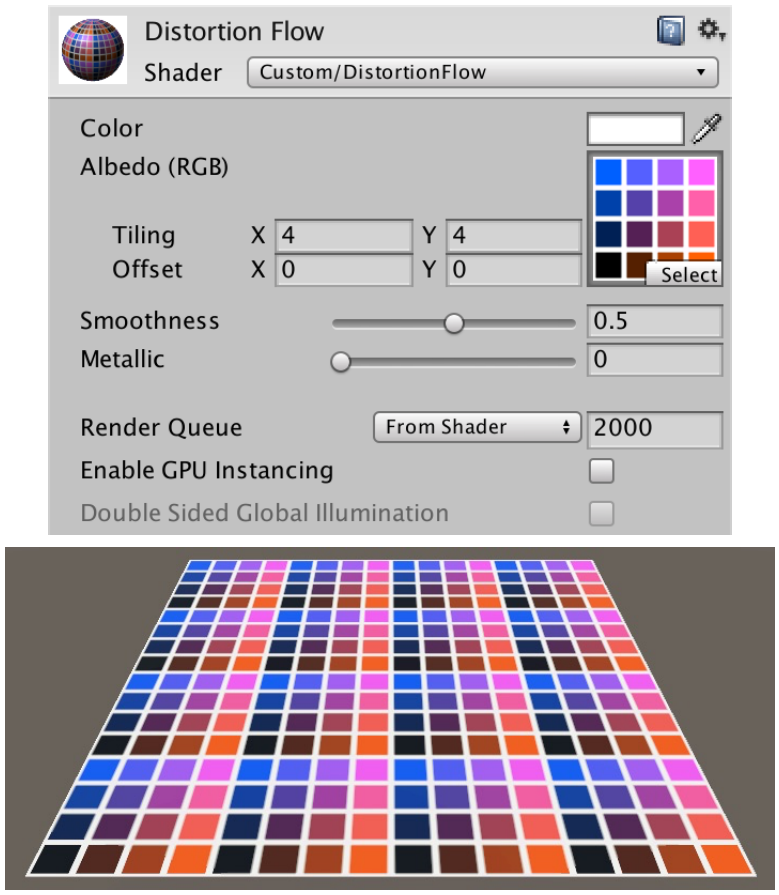
For this tutorial, you can start with a new project, set to use linear color space rendering. If you're using Unity 2018, select the default 3D pipeline, not lightweight or HD. Then create a new standard surface shader. As we're going to simulate a flowing surface by distorting texture mapping, name it *DistortionFlow*. Below is the new shader, with all comments and unneeded parts removed.

```
Shader "Custom/DistortionFlow" {
    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 Standard fullforwardshadows
        #pragma target 3.0

        sampler2D _MainTex;

        struct Input {
            float2 uv_MainTex;
        };

        half _Glossiness;
        half _Metallic;
        fixed4 _Color;

        void surf (Input IN, inout SurfaceOutputStandard o) {
            fixed4 c = tex2D(_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            o.Metallic = _Metallic;
            o.Smoothness = _Glossiness;
            o.Alpha = c.a;
        }
        ENDCG
    }

    FallBack "Diffuse"
}
```

To make it easy to see how the UV coordinates are deformed, you can use this test texture.

UV test texture.

Create a material that uses our shader, with the test texture as its albedo map. Set its tiling to 4 so we can see how the texture repeats. Then add a quad to the scene with this material. For best viewing, rotate it 90° around its X axis so it lies flat in the XZ plane. That makes it easy to look at it from any angle.



Distortion Flow material on a quad.

1.2 Flowing UV

The code for flowing UV coordinates is generic, so we'll put it in a separate *Flow.cginc* include file. All it needs to contain is a `FlowUV` function that has a UV and a time parameter. It should return the new flowed UV coordinates. We begin with the most straightforward displacement, which is simply adding the time to both coordinates.

```
#if !defined(FLOW_INCLUDED)
#define FLOW_INCLUDED

float2 FlowUV (float2 uv, float time) {
	return uv + time;
}

#endif
```

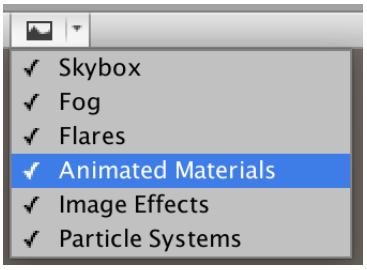
Include this file in our shader and invoke `FlowUV` with the main texture coordinates and the current time, which Unity makes available via `_Time.y`. Then use the new UV coordinates to sample our texture.

```
		#include "Flow.cginc"

		sampler2D _MainTex;

		…

		void surf (Input IN, inout SurfaceOutputStandard o) {
			float2 uv = FlowUV(IN.uv_MainTex, _Time.y);
			fixed4 c = tex2D(_MainTex, uv) * _Color;
			o.Albedo = c.rgb;
			o.Metallic = _Metallic;
			o.Smoothness = _Glossiness;
			o.Alpha = c.a;
		}
```

Diagonally sliding UV.

As we're increasing both coordinates by the same amount, the texture slides diagonally. Because we adding the time, it slides from top right to bottom left. And because we're using the default wrap mode for our texture, the animation loops every second.

The animation is only visible when the time value increases. This is the case when the editor is in play mode, but you can also enable time progression in edit mode, by enabling *Animated Materials* via the *Scene* window toolbar.



Animated Materials enabled.

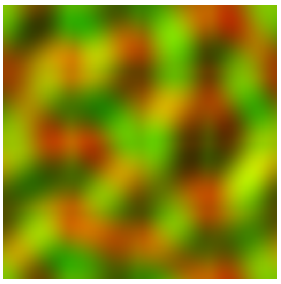
Actually, the time value used by materials increases each time the editor redraws the scene. So when *Animated Materials* is disabled you will see the texture slide a bit each time you edit something. *Animated Materials* just forces the editor to redraw the scene all the time. So only turn it on when you need it.

1.3 Flow Direction

Instead of always flowing in the same direction, you can use a velocity vector to control the direction and speed of the flow. You could add this vector as a property to the material. However, then we're still limited to using the same vector for the entire material, which looks like a rigid sliding surface. To make make something look like flowing liquid, it has to locally change over time besides moving in general.

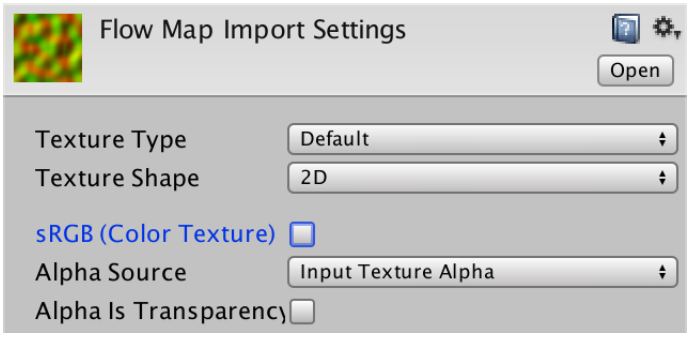
We could get rid of the static appearance by adding another velocity vector, using that to sample the texture a second time, and combining both samples. When using two slightly different vectors, we end up with a morphing texture. However, we're still limited to flowing the entire surface the same way. This is often sufficient for open water or straight flows, not not in more complex situations.

To support more interesting flows, we must somehow vary the flow vector across the surface of our material. The most straightforward way to do this is via a flow map. This is a texture that contains 2D vectors. Here is such a texture, with the vector's U component in the R channel and the V component in the G channel. It doesn't need to be large, because we don't need sharp sudden changes and we can rely on bilinear filtering to keep it smooth.



Flow map.

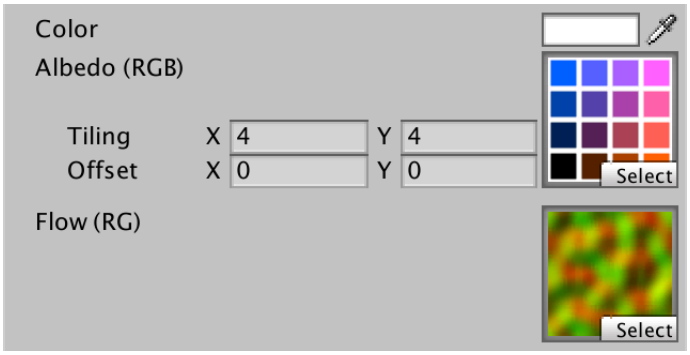
This texture was created with curl noise, which is explained in the Noise Derivatives tutorial, but the details of its creation don't matter. It contains multiple clockwise and counterclockwise rotating flows, without any sources or sinks. Make sure that it is imported as a regular 2D texture that isn't sRGB, as it doesn't contain color data.



Import as non-sRGB texture.

Add a property for the flow map to our material. It doesn't need a separate UV tiling and offset, so give it the `NoScaleOffset` attribute. The default is that there is no flow, which corresponds to a black texture.

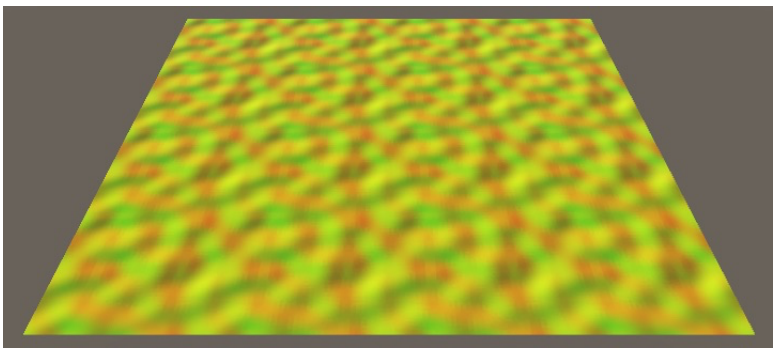
```
		_MainTex ("Albedo (RGB)", 2D) = "white" {}
		[NoScaleOffset] _FlowMap ("Flow (RG)", 2D) = "black" {}
```



Material with flow map.

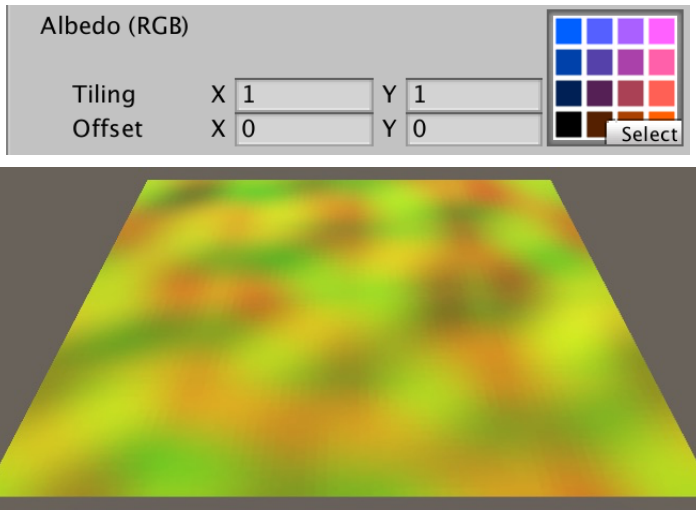
Add a variable for the flow map and sample it to get the flow vector. Then temporarily visualize it by using it as the albedo.

```
		sampler2D _MainTex, _FlowMap;

		…

		void surf (Input IN, inout SurfaceOutputStandard o) {
			float2 flowVector = tex2D(_FlowMap, IN.uv_MainTex).rg;
			float2 uv = FlowUV(IN.uv_MainTex, _Time.y);
			fixed4 c = tex2D(_MainTex, uv) * _Color;
			o.Albedo = c.rgb;
			o.Albedo = float3(flowVector, 0);
			…
		}
```



Tiled flow vectors.

The texture appears brighter in the scene, because it's linear data. That's fine, because we're not supposed to use it as a color anyway. As the main UV coordinates of the surface shader use the tiling and offset of the main texture, our flow map gets tiled as well. We don't need a tiling flow map, so set the material's tiling back to 1.



Flow vectors without tiling.

1.4 Directed Sliding

Now that we have flow vectors, we can add support for them to our `FlowUV` function. Add a parameter for them, then multiply them with the time before subtracting from the original UV. We subtract because that makes the flow go in the direction of the vector.

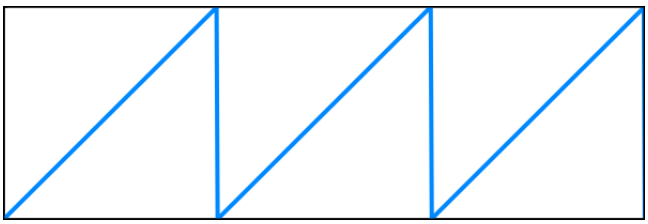
```
float2 FlowUV (float2 uv, float2 flowVector, float time) {
	return uv - flowVector * time;
}
```

Pass the flow vector to the function, but before doing so make sure that the vector is valid. Like with a normal map, the vector can point in any direction, so can contain negative components. Therefore, the vector is encoded the same way as in a normal map. We have to manually decode it. Also, revert to the original albedo.

```
			float2 flowVector = tex2D(_FlowMap, IN.uv_MainTex).rg * 2 - 1;
			float2 uv = FlowUV(IN.uv_MainTex, flowVector, _Time.y);
			fixed4 c = tex2D(_MainTex, uv) * _Color;
			o.Albedo = c.rgb;
//			o.Albedo = float3(flowVector, 0);
```

too much distortion.

We quickly end up with a texture that is way too distorted. This happens because the texture gets moved in multiple directions, stretching and squashing it more and more as time progresses. To prevent it from turning into a mess, we have to reset the animation at some point. The simplest way to do this is by only using the fractional part of the time for the animation. Thus, it progresses from 0 up to 1 as normal, but then resets to 0, forming a sawtooth pattern.



Sawtooth progress.

As this is particular to the flow animation and not time in general, create the sawtooth progression in `FlowUV`.

```
float2 FlowUV (float2 uv, float2 flowVector, float time) {
	float progress = frac(time);
	return uv - flowVector * progress;
}
```

Resetting the progression each second.

We can now see that the texture indeed gets deformed in different directions and at different speeds. Besides the sudden reset, what is most obvious is that the texture quickly becomes blocky as its deformation increases. This is caused by the compression of the flow map. The default compression setting uses the DXT1 format, which is where the blockiness comes from. These artifacts are typically not obvious when using organic textures, but are glaring when deforming clear patterns, like our test texture. So I've used an uncompressed flow map for all screenshots and movies in this tutorial.

Without compression.

Why not just use a higher resolution flow map?

While that is possible, flow maps often cover large areas and thus end up with low effective resolution. As long as you're not using extreme deformation, there is no problem. The deformations shown in this tutorial are very strong, to make them visually obvious.

2 Seamless Looping

At this point we can animate a nonuniform flow, but it resets each second. To make it loop without discontinuity, we have to somehow get the UV back to their original values, before distortion. Time only goes forward, so we cannot rewind the distortion. Trying that would result in a flow that goes back and forth instead of in a consistent direction. We have to find another way.

2.1 Blend Weight

We cannot avoid resetting the progression of the distortion, but we can try to hide it. What we could do is fade the texture to black as we approach maximum distortion. If we also start with black and fade in the texture at the start, then the sudden reset happens when the entire surface is black. While this is very obvious, at least there is no sudden visual discontinuity.

To make the fading possible, let's add a blend weight to the output of our `FlowUV` function, renaming it to `FlowUVW`. The weight is put in the third component, which has effectively been 1 up to now, so let's start with that.

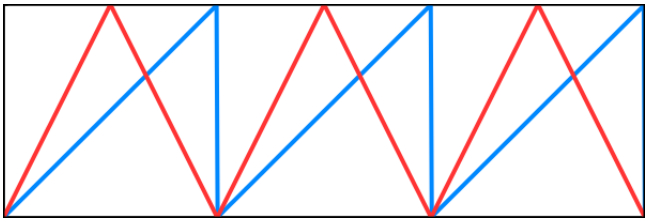
```
float3 FlowUVW (float2 uv, float2 flowVector, float time) {
	float progress = frac(time);
	float3 uvw;
	uvw.xy = uv - flowVector * progress;
	uvw.z = 1;
	return uvw;
}
```

We can fade the texture by multiplying it with the weight that is now available to our shader.

```
			float3 uvw = FlowUVW(IN.uv_MainTex, flowVector, _Time.y);
			fixed4 c = tex2D(_MainTex, uvw.xy) * uvw.z * _Color;
```

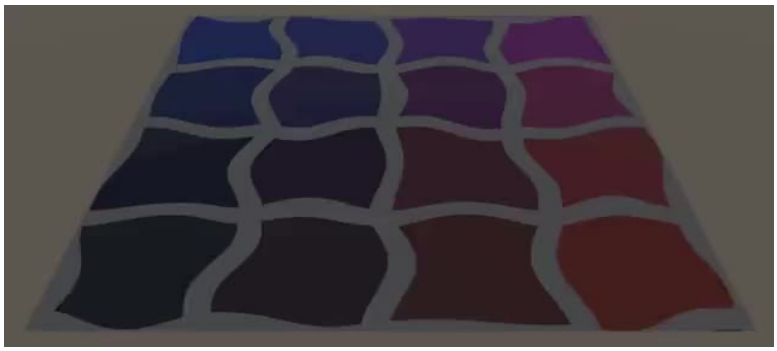
2.2 Seesaw

Now we must create a weight function $w(p)$ where $w(0) = w(1) = 0$. And halfway it should reach full strength, so $w\left(\frac{1}{2}\right) = 1$. The simplest function that matches these criteria is a triangle wave, $w(p) = 1 - |1 - 2p|$. Use that for our weight.



Sawtooth with matching triangle wave.

```
    uvw.z = 1 - abs(1 - 2 * progress);
```



Triangle wave modulation.

Why not use a smoother function?

You could also use a sine wave or apply the smoothstep function. But those functions make the shader more complex, while not affecting the final result much. A triangle wave is sufficient.

2.3 Time Offset

While technically we have removed the visual discontinuity, we have introduced a black pulsing effect. The pulsing is very obvious because it happens everywhere at once. It might be less obvious if we could spread it out over time. We can do this by offsetting the time by a varying amount across the surface. Some low-frequency Perlin noise is very suitable for this. Instead of adding another texture, we'll pack the noise in our flow map. Here is the same flow map as before, but now with noise in its A channel. The noise is unrelated to the flow vectors.

Flow map with noise in A channel.

To indicate that we expect noise in the flow map, update its label.

```
		[NoScaleOffset] _FlowMap ("Flow (RG, A noise)", 2D) = "black" {}
```

Sample the noise and add it to the time before passing it to `FlowUVW`.

```
			float2 flowVector = tex2D(_FlowMap, IN.uv_MainTex).rg * 2 - 1;
			float noise = tex2D(_FlowMap, IN.uv_MainTex).a;
			float time = _Time.y + noise;
			float3 uvw = FlowUVW(IN.uv_MainTex, flowVector, time);
```

Time with offset.

Why sample the flow map twice?

Just to point out that the shader compiler will optimize that into a single texture sample.

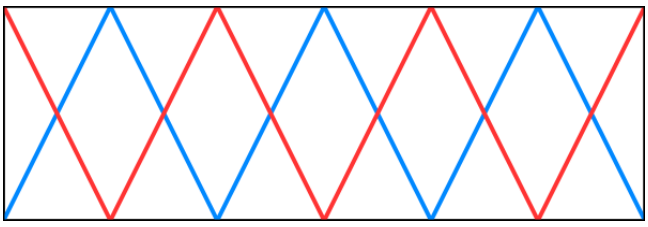
The black pulse is still there, but it has changed into a wave that spreads across the surface in an organic way. This is much easier to obfuscate than uniform pulsing. As a bonus, the time offset also made the progression of the distortion nonuniform, resulting in a more varied distortion overall.

2.4 Combining Two Distortions

Instead of fading to black, we could blend with something else, for example the original undistorted texture. But then we would see a fixed texture fade in and out, which would destroy the illusion of flow. We can solve that by blending with another distorted texture. This requires us to sample the texture twice, each with different UVW data.

So we end up with two pulsing patterns, A and B. When A's weight is 0, B's should be 1, and vice versa. That way the black pulse is hidden. This is done by shifting the phase of B by half its period, which means adding 0.5 to its time. But that's a detail of how `FlowUVW` works, so let's just add a boolean parameter to indicate whether we want the UVW for the A or B variant.

```
float3 FlowUVW (float2 uv, float2 flowVector, float time, bool flowB) {
	float phaseOffset = flowB ? 0.5 : 0;
	float progress = frac(time + phaseOffset);
	float3 uvw;
	uvw.xy = uv - flowVector * progress;
	uvw.z = 1 - abs(1 - 2 * progress);
	return uvw;
}
```



Weights of A and B always sum to 1.

We now have to invoke `FlowUVW` twice, once with `false` and once with `true` as its last argument. Then sample the texture twice, multiply both with their weights, and add them to arrive at the final albedo.

```
		float time = _Time.y + noise;

		float3 uvwA = FlowUVW(IN.uv_MainTex, flowVector, time, false);
		float3 uvwB = FlowUVW(IN.uv_MainTex, flowVector, time, true);

		fixed4 texA = tex2D(_MainTex, uvwA.xy) * uvwA.z;
		fixed4 texB = tex2D(_MainTex, uvwB.xy) * uvwB.z;

		fixed4 c = (texA + texB) * _Color;
```

Blending two phases.

The black pulsing wave is no longer visible. The wave is still there, but now forms the transition between the two phases, which is far less obvious.

A side effect of blending between two patterns offset by half their period is that our animation's duration has been halved. It now loops twice per second. But we don't have to use the same pattern twice. We can offset the UV coordinates of B by half a unit. This makes the patterns different—while using the same texture—without introducing any directional bias.

```
	uvw.xy = uv - flowVector * progress + phaseOffset;
```

Different UV for A and B.

Because we use a regular test pattern, the white grid lines of A and B overlap. But the colors of their squares are different. As a result, the final animation alternates between two color configurations, and again takes a second to repeat.

2.5 Jumping UV

Besides always offsetting the UV of A and B by half a unit, it is also possible to offset the UV per phase. That will cause the animation to change over time, so it takes longer before it loops back to the exact same state.

We could simply slide the UV coordinates based on time, but that would cause the whole animation to slide, introducing a directional bias. We can avoid visual sliding by keeping the UV offset constant during each phase, and jumping to a new offset between phases. In other words, we make the UV jump each time the weight is zero. This is done by adding some jump offset to the UV, multiplied by the integer portion of the time. Adjust `FlowUVW` to support this, with a new parameter to specify the jump vector.

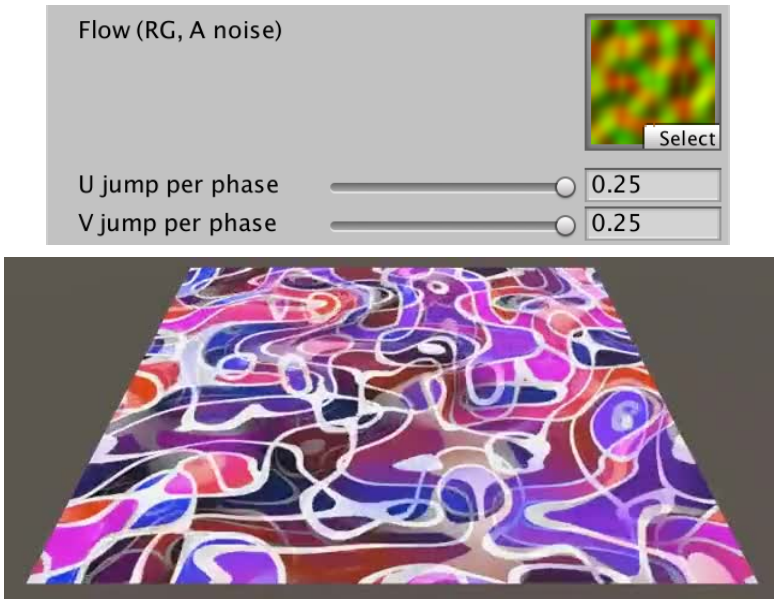
```
float3 FlowUVW (
	float2 uv, float2 flowVector, float2 jump, float time, bool flowB
) {
	float phaseOffset = flowB ? 0.5 : 0;
	float progress = frac(time + phaseOffset);
	float3 uvw;
	uvw.xy = uv - flowVector * progress + phaseOffset;
	uvw.xy += (time - progress) * jump;
	uvw.z = 1 - abs(1 - 2 * progress);
	return uvw;
}
```

Add two parameters to our shader to control the jump. We use two floats instead of a single vector, so we can use range sliders. Because we're blending between two patterns that are offset by half, our animation already contains the UV offset sequence $0 \rightarrow \frac{1}{2}$ per phase. The jump offset gets added on top of this. This means that if we were to jump by half, the progression would become $0 \rightarrow \frac{1}{2} \rightarrow \frac{1}{2} \rightarrow 0$ over two phases, which is not what we want. We should jump by a quarter at most, which produces $0 \rightarrow \frac{1}{2} \rightarrow \frac{1}{4} \rightarrow \frac{3}{4} \rightarrow \frac{1}{2} \rightarrow 0 \rightarrow \frac{3}{4} \rightarrow \frac{1}{4}$ over four phases. A negative offset of at most a quarter is also possible. That would produce the sequence $0 \rightarrow \frac{1}{2} \rightarrow \frac{3}{4} \rightarrow \frac{1}{4} \rightarrow \frac{1}{2} \rightarrow 0 \rightarrow \frac{1}{4} \rightarrow \frac{3}{4}$.

```
		[NoScaleOffset] _FlowMap ("Flow (RG, A noise)", 2D) = "black" {}
		_UJump ("U jump per phase", Range(-0.25, 0.25)) = 0.25
		_VJump ("V jump per phase", Range(-0.25, 0.25)) = 0.25
```

Add the required float variables to our shader, use them to construct the jump vector, and pass it to `FlowUVW`.

```
		sampler2D _MainTex, _FlowMap;
		float _UJump, _VJump;

		…

		void surf (Input IN, inout SurfaceOutputStandard o) {
			float2 flowVector = tex2D(_FlowMap, IN.uv_MainTex).rg * 2 - 1;
			float noise = tex2D(_FlowMap, IN.uv_MainTex).a;
			float time = _Time.y + noise;
			float2 jump = float2(_UJump, _VJump);

			float3 uvwA = FlowUVW(IN.uv_MainTex, flowVector, jump, time, false);
			float3 uvwB = FlowUVW(IN.uv_MainTex, flowVector, jump, time, true);

			…
		}
```

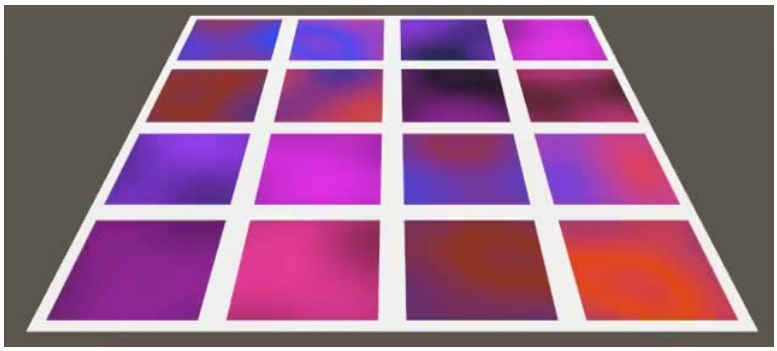


Material with maximum jump.

At maximum jump we end up with a sequence of eight UV offsets before it repeats. As we go through two offsets per phase and each phase is one second long, our animation now loops every four seconds.

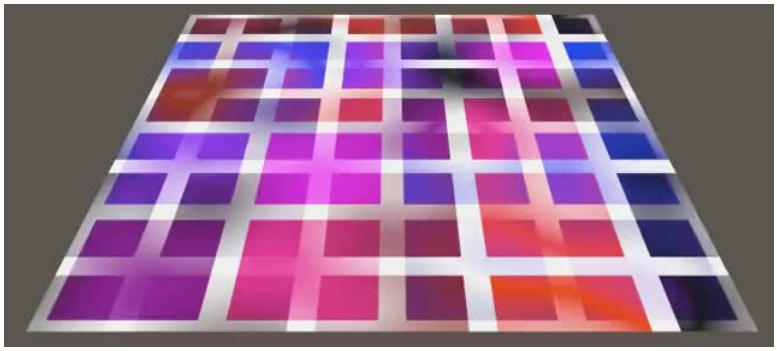
2.6 Analyzing Jumps

To better see how UV jumping works, you can set the flow vectors to zero so you can focus on the offsets. First, consider the animation without any jump, just the original alternating patterns.



Jump 0, duration 1s.

You can see that each square is alternating between two colors. You can also see that we're alternating between the same texture offset by half, but this is not immediately obvious and there is no directional bias. Next, look at the animation with maximum jump in both dimensions.



Jump 0.25, duration 4s.

The result looks different, because jumping by a quarter causes the grid lines of our test texture to move, alternating between squares and crosses. The white lines still do not show a directional bias, but the colored squares now do. The pattern moves diagonally, but not in an immediately obvious way. It takes half a step forward, then a quarter step back, repeat. Had we used the minimum jump of −0.25, then it would take half a step forward, followed by a quarter step forward, repeat. To make the directional bias more obvious, use a jump that isn't symmetrical, for example 0.2.

Jump 0.2, duration 2.5s.

In this case, the white grid lines also appear to move. But because we're still using a large jump that is fairly close to symmetrical, the movement can be interpreted to go in multiple directions, depending on how you focus on the image. If you change your focus, you can easily lose track of the direction you thought it was flowing.

Because we're using a jump of 0.2, the animation repeats after five phases, so five seconds. However, because we blend between two offset phases there is a potential crossover point in the middle of each phase. If the animation would loop after an odd number of phases, it actually loops twice as the phases cross halfway. So in this case the duration is only 2.5s.

You don't have to jump U and V by the same amount. Besides changing the nature of the directional bias, using different jump values per dimension also affects the loop duration. For example, consider an U jump of 0.25 and a V jump of 0.1. U loops every four cycles, while V loops every ten. So after four cycles U has looped, but V hasn't yet, so the animation hasn't completed a loop either. Only when both U and V complete a cycle at the end of the same phase do we reach the end of the animation. When using rational numbers for the jumps, the loop duration is equal to the least common multiple of their denominators. In the case of 0.25 and 0.1, that's 4 and 10, for which the least common multiple is 20.

There is no obvious way to pick a jump vector so you end up with a long loop duration. For example, if we use 0.25 and 0.2 instead of 0.25 and 0.1, do we get a longer or shorter duration? As the least common multiple of 4 and 5 is also 20, the duration is the same. Also, while you could come up with values that theoretically take a long time or even forever to loop, most aren't practically useful. We cannot perceive changes that are too small, plus there's numerical precision limitations, which can cause theoretically good jump values to appear either unchanging under casual observation, or to loop much quicker than expected.

I think good jump values—besides zero—sit somewhere between 0.2 and 0.25, either positive or negative. I've come up with $\frac{6}{25} = 0.24$ and $\frac{5}{24} \approx 0.2083333$ as a nice simple pair that fits the criteria. The first value completes six jump cycles after 25 phases, while the second completes five cycles after 24 phases. The total theoretical loop takes 600 phases, which is ten minutes at the speed of one phase per second.

I'll leave the jump values at zero for the rest of this tutorial, just so I can keep the looping animations short.

3 Animation Tweaks

Now that we have a basic flow animation, let's add some more configuration options to it, so we can fine-tune its appearance.

3.1 Tiling

First, let's make it possible to tile the texture that gets distorted. We cannot rely on the main tiling and offset of the surface shader, because that also affects the flow map. Instead, we need a separate tiling property for the texture. It typically only makes sense to distort a square texture, so we only need a single tiling value.

To keep the flow the same regardless of the tiling, we have to apply it to the UV after flowing, but before adding the offset for phase B. So it has to be done in `FlowUVW`, which means that our function needs a tiling parameter.

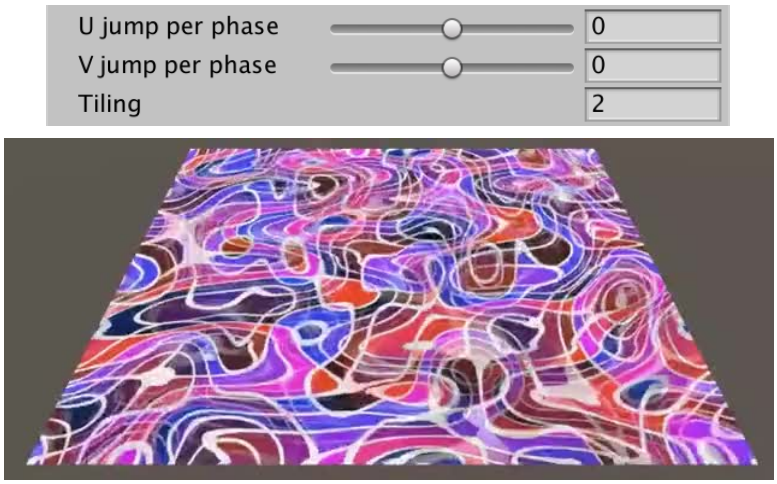
```
float3 FlowUVW (
	float2 uv, float2 flowVector, float2 jump,
	float tiling, float time, bool flowB
) {
	…
//	uvw.xy = uv - flowVector * progress + phaseOffset;
	uvw.xy = uv - flowVector * progress;
	uvw.xy *= tiling;
	uvw.xy += phaseOffset;
	…
}
```

Add a tiling property to our shader as well, with 1 as the default value.

```
		_UJump ("U jump per phase", Range(-0.25, 0.25)) = 0.25
		_VJump ("V jump per phase", Range(-0.25, 0.25)) = -0.25
		_Tiling ("Tiling", Float) = 1
```

Then add the needed variable and pass it to `FlowUVW`.

```
		float _UJump, _VJump, _Tiling;

		…

		void surf (Input IN, inout SurfaceOutputStandard o) {
			…

			float3 uvwA = FlowUVW(
				IN.uv_MainTex, flowVector, jump, _Tiling, time, false
			);
			float3 uvwB = FlowUVW(
				IN.uv_MainTex, flowVector, jump, _Tiling, time, true
			);

			…
		}
```



Tiling set to 2, duration still 1s.

When tiling is set to 2, the animation appears to flow twice as fast as before. But that's just because the texture has been scaled. The animation still takes one second to loop, when not jumping the UV.

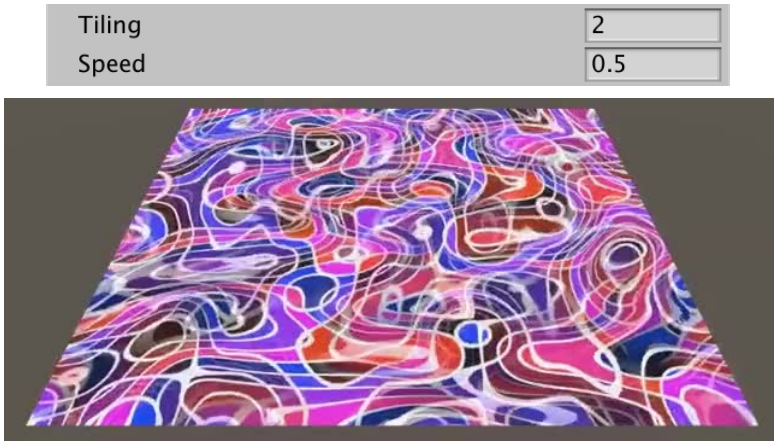
3.2 Animation Speed

The speed of the animation can be directly controlled by scaling the time. This affects the entire animation, so also its duration. Add a speed shader property to support this.

```
		_Tiling ("Tiling", Float) = 1
		_Speed ("Speed", Float) = 1
```

Simply multiply `_Time.y` by the corresponding variable. The noise value should be added afterwards, so the time offset remains unaffected.

```
		float _UJump, _VJump, _Tiling, _Speed;

		…

		void surf (Input IN, inout SurfaceOutputStandard o) {
			float2 flowVector = tex2D(_FlowMap, IN.uv_MainTex).rg * 2 - 1;
			float noise = tex2D(_FlowMap, IN.uv_MainTex).a;
			float time = _Time.y * _Speed + noise;
			…
		}
```



Speed set to 0.5, duration now 2s.

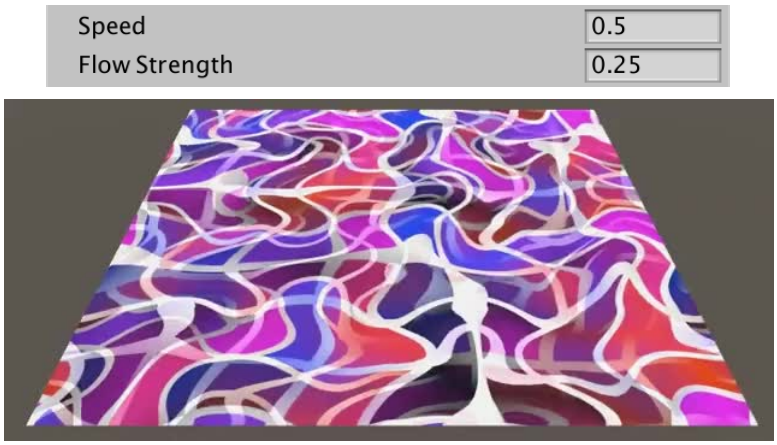
3.3 Flow Strength

The velocity of the flow is dictated by the flow map. We can speed it up or slow it down by adjusting the animation speed, but that also affects the phase length and animation duration. Another way to change the apparent flow speed is by scaling the flow vectors. By adjusting the strength of the flow we can speed it up, slow it down, or even reverse it, without affecting time. This also changes the amount of distortion. Add a *Flow Strength* shader property to make this possible.

```
		_Speed ("Speed", Float) = 1
		_FlowStrength ("Flow Strength", Float) = 1
```

Simply multiply the flow vector by the corresponding variable before using it.

```
		float _UJump, _VJump, _Tiling, _Speed, _FlowStrength;

		…

		void surf (Input IN, inout SurfaceOutputStandard o) {
			float2 flowVector = tex2D(_FlowMap, IN.uv_MainTex).rg * 2 - 1;
			flowVector *= _FlowStrength;
			…
		}
```



Flow strength set to 0.25, duration still 2s.

3.4 Flow Offset

Another possible tweak is to control where the animation starts. Up to this point we've always started at zero distortion at the beginning of each phase, progressing to maximum distortion. As the phase's weight reaches 1 at the halfway point, the pattern is most clear when the distortion is at half strength. Thus, we mostly see a half-distorted texture. This configuration is often fine, but not always. For example, in Portal 2 the floating debris texture is mostly seen in its undistorted state. This is done by offsetting the flow by −0.5 when distorting the UV coordinates.

Let's support this too, by adding a `flowOffset` parameter to `FlowUVW`. Add it to the progress when multiplying with the flow vector only.

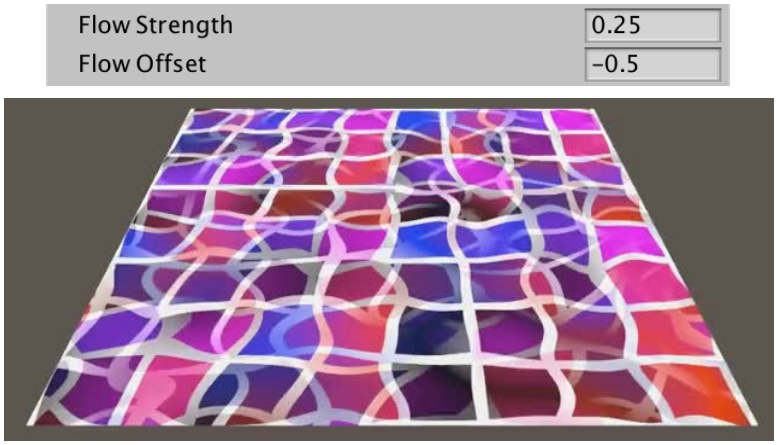
```
float3 FlowUVW (
	float2 uv, float2 flowVector, float2 jump,
	float flowOffset, float tiling, float time, bool flowB
) {
	float phaseOffset = flowB ? 0.5 : 0;
	float progress = frac(time + phaseOffset);
	float3 uvw;
	uvw.xy = uv - flowVector * (progress + flowOffset);
	uvw.xy *= tiling;
	uvw.xy += phaseOffset;
	uvw.xy += (time - progress) * jump;
	uvw.z = 1 - abs(1 - 2 * progress);
	return uvw;
}
```

Next, add a property to control the flow offset the shader. Its practical values are 0 and −0.5, but you can experiment with other values as well.

```
		_FlowStrength ("Flow Strength", Float) = 1
		_FlowOffset ("Flow Offset", Float) = 0
```

Pass the corresponding variable to `FlowUVW`.

```
		float _UJump, _VJump, _Tiling, _Speed, _FlowStrength, _FlowOffset;

		…

		void surf (Input IN, inout SurfaceOutputStandard o) {
			…

			float3 uvwA = FlowUVW(
				IN.uv_MainTex, flowVector, jump,
				_FlowOffset, _Tiling, time, false
			);
			float3 uvwB = FlowUVW(
				IN.uv_MainTex, flowVector, jump,
				_FlowOffset, _Tiling, time, true
			);

			…
		}
```



Flow offset set to −0.5.

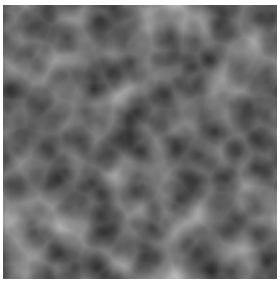
With a flow offset of −0.5 there is no distortion at the peak of each phase. But the overall result is still distorted, due to the time offset.

4 Texturing

Our distortion flow shader is now fully functional. Let's see how it looks with something else than the test texture that we've been using so far.

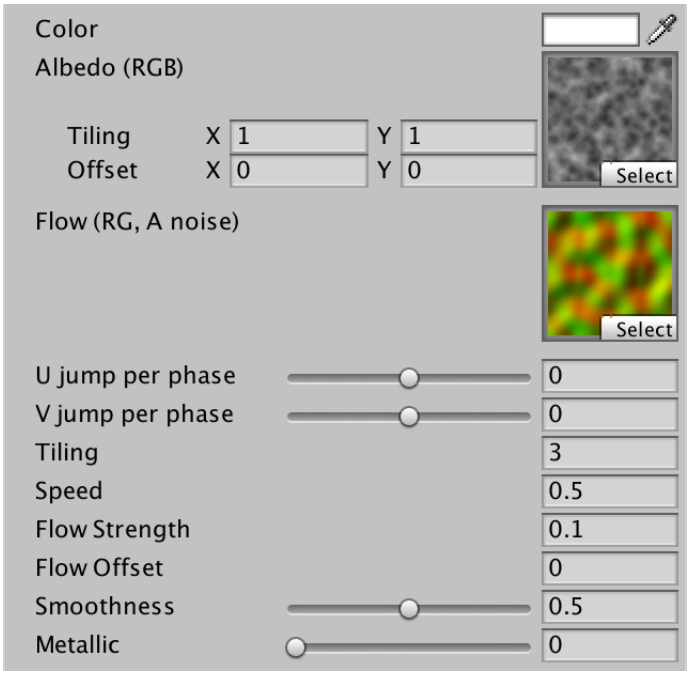
4.1 Abstract Water

The most common use of the distortion effect is to simulate a water surface. But because the distortion can be in any direction we cannot use a texture that suggests a specific flow direction. It's not really possible to make correct waves without suggesting a direction, but we don't need to be realistic. It just has to look like water when the texture is distorted and blended. For example, here is a simple noise texture that combines one octave of low-frequency Perlin and Voronoi noise. It's an abstract grayscale representation of water, dark at the bottom and light at the top of waves.



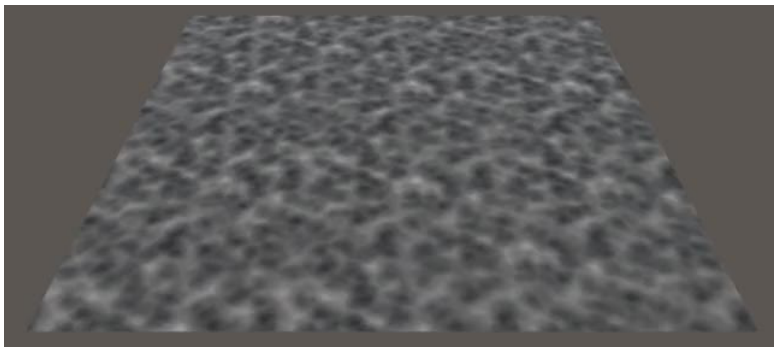
Water texture.

Use this texture for the albedo map of our material. Besides that, I've used no jump, a tiling of 3, speed of 0.5, flow strength of 0.1, and no flow offset.



Flowing water.

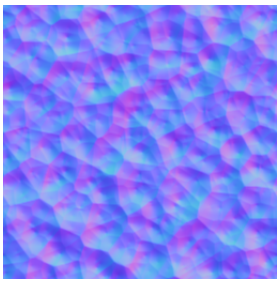
Even though the noise texture by itself doesn't really look like water, the distorted and animated result is starting to look like it. You can also check how it would look without distortion, by temporarily setting the flow strength to zero. This would represent stationary water, and it should look at least somewhat acceptable.



Stationary water.

4.2 Normal Map

The albedo map is only a preview, as flowing water is mostly defined by the way its surface changes vertically, which alters how it interacts with light. We need a normal map for that. Here is one, created by interpreting the albedo texture as a height map, but with the heights scaled by 0.1 so the effect isn't too strong.



Normal map.

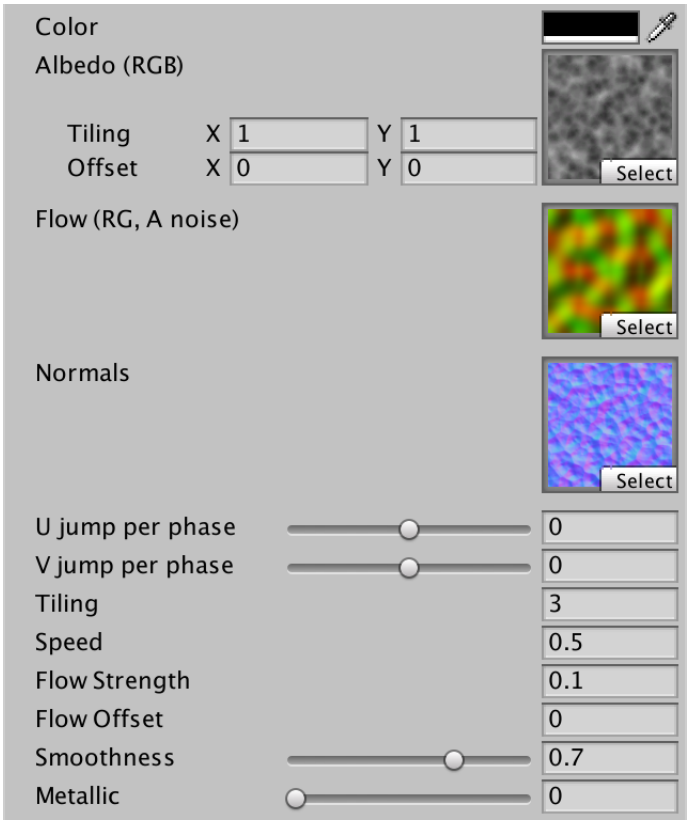
Add a shader property for the normal map.

```
		[NoScaleOffset] _FlowMap ("Flow (RG, A noise)", 2D) = "black" {}
		[NoScaleOffset] _NormalMap ("Normals", 2D) = "bump" {}
```

Sample the normal map for both A and B, apply their weights, and use their normalized sum as the final surface normal.

```
		sampler2D _MainTex, _FlowMap, _NormalMap;
		…

		void surf (Input IN, inout SurfaceOutputStandard o) {
			…

			float3 normalA = UnpackNormal(tex2D(_NormalMap, uvwA.xy)) * uvwA.z;
			float3 normalB = UnpackNormal(tex2D(_NormalMap, uvwB.xy)) * uvwB.z;
			o.Normal = normalize(normalA + normalB);

			fixed4 texA = tex2D(_MainTex, uvwA.xy) * uvwA.z;
			fixed4 texB = tex2D(_MainTex, uvwB.xy) * uvwB.z;

			…
		}
```

Add the normal map to our material. Also increase its smoothness to something like 0.7, then change the light so you get plenty of specular reflections. I kept the view the same but rotated the directional light 180° to (50, 150, 0). Also set albedo to black, so we only see the effect of animating normals.



Flowing water.

The distorted and animated normal map creates a pretty convincing illusion of flowing water. But how does it hold up when the flow strength is zero?

Stationary water.

At first glance it might look fine, but if you focus on specific highlights it quickly becomes obvious that they alternate between two states. Fortunately, this can be solved by using jump values other than zero.

Maximum jump, speed set to 1.

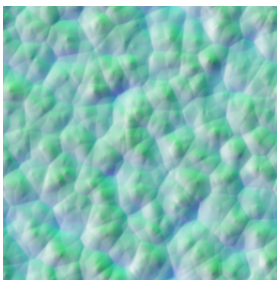
4.3 Derivative Map

Although the resulting normals look good, averaging normals doesn't make much sense. As explained in Rendering 6, Bumpiness, the correct approach would be to convert the normal vectors to height derivatives, add them, and then convert back to a normal vector. This is especially true for waves that travel across a surface.

As we're typically using DXT5nm compression for our normal maps, we first have to reconstruct the Z component of both normals—which requires a square root computation—then convert to derivatives, combine, and normalize. But we don't need the original normal vectors, so we could also skip the conversion by storing the derivatives in a map, instead of the normals.

A derivative map works just like a normal map, except it contains the height derivatives in the X and Y dimensions. However, without extra scaling the derivative map can only support surface angles up to 45°, because the derivative of that is 1. As you typically won't use such steep waves, that limitation is acceptable.

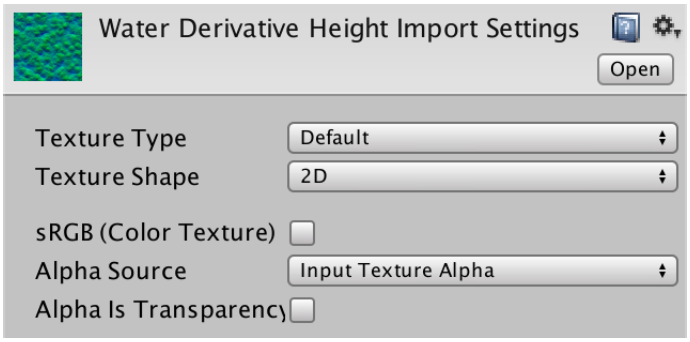
Here is a derivative map describing the same surface as the earlier normal map, with the X derivative stored in the A channel and the Y derivative stored in the G channel, just like a normal map. As a bonus, it also contains the original height map in its B channel. But again the derivatives are calculated by scaling the height by 0.1.



Derivative plus height map.

> **Why not store the height scaled by 0.1 too?**
>
> The height data is stored at full strength to minimize loss of precision.

As the texture isn't a normal map, import it as a regular 2D texture. Make sure to indicate that it is not an sRGB texture.



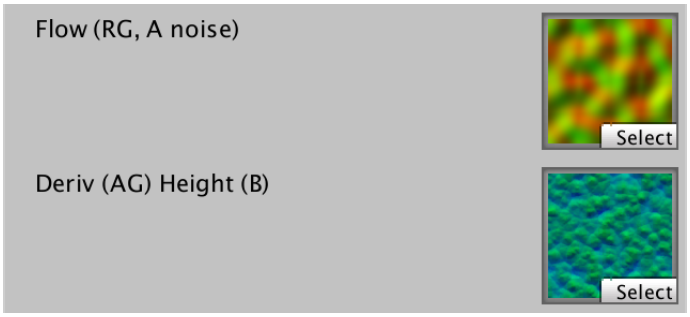
Import settings.

Replace the normal map shader property with one for our derivate-plus-height map.

```
//		[NoScaleOffset] _NormalMap ("Normals", 2D) = "bump" {}
		[NoScaleOffset] _DerivHeightMap ("Deriv (AG) Height (B)", 2D) = "black" {}
```

Replace the shader variable, sampling, and normal construction as well. We cannot use `UnpackNormal` anymore, so create a custom `UnpackDerivativeHeight` function that puts the correct data channels in a float vector and decodes the derivatives.

```
		sampler2D _MainTex, _FlowMap, _DerivHeightMap;
		…

		float3 UnpackDerivativeHeight (float4 textureData) {
			float3 dh = textureData.agb;
			dh.xy = dh.xy * 2 - 1;
			return dh;
		}

		void surf (Input IN, inout SurfaceOutputStandard o) {
			…
//			float3 normalA = UnpackNormal(tex2D(_NormalMap, uvwA.xy)) * uvwA.z;
//			float3 normalB = UnpackNormal(tex2D(_NormalMap, uvwB.xy)) * uvwB.z;
//			o.Normal = normalize(normalA + normalB);

			float3 dhA =
				UnpackDerivativeHeight(tex2D(_DerivHeightMap, uvwA.xy)) * uvwA.z;
			float3 dhB =
				UnpackDerivativeHeight(tex2D(_DerivHeightMap, uvwB.xy)) * uvwB.z;
			o.Normal = normalize(float3(-(dhA.xy + dhB.xy), 1));

			…
		}
```



With derivative map instead of normal map.

The resulting surface normals look almost the same as when using the normal map, they're just cheaper to compute. As we now also have access to the height data, we could use this to colorize the surface as well. This can be useful for debugging, so let's temporarily replace the original albedo.

```
			o.Albedo = c.rgb;
			o.Albedo = dhA.z + dhB.z;
```

Using height as albedo.

The surface appears lighter than when using the albedo texture, even through both contain the same height data. It's different because we're now using linear data, while the albedo texture is interpreted as sRGB data. To get the same result, we would have to manually convert the height data from gamma to linear color space. We can approximate this by simply squaring it.

```
		o.Albedo = pow(dhA.z + dhB.z, 2);
```

Using squared height.

4.4 Height Scale

Another benefit of working with derivatives instead of normal vectors is that they can be easily scaled. The derived normals will match the adjusted surface. This makes it possible to correctly scale the height of the waves. Let's add a height scale property to our shader to support this.

```
		_FlowOffset ("Flow Offset", Float) = 0
		_HeightScale ("Height Scale", Float) = 1
```

All we need to do is factor the height scale into the sampled derivative plus height data.

```
        float _HeightScale;

        …

        void surf (Input IN, inout SurfaceOutputStandard o) {
            …

            float3 dhA =
                UnpackDerivativeHeight(tex2D(_DerivHeightMap, uvwA.xy)) *
                (uvwA.z * _HeightScale);
            float3 dhB =
                UnpackDerivativeHeight(tex2D(_DerivHeightMap, uvwB.xy)) *
                (uvwB.z * _HeightScale);
            …
        }
```

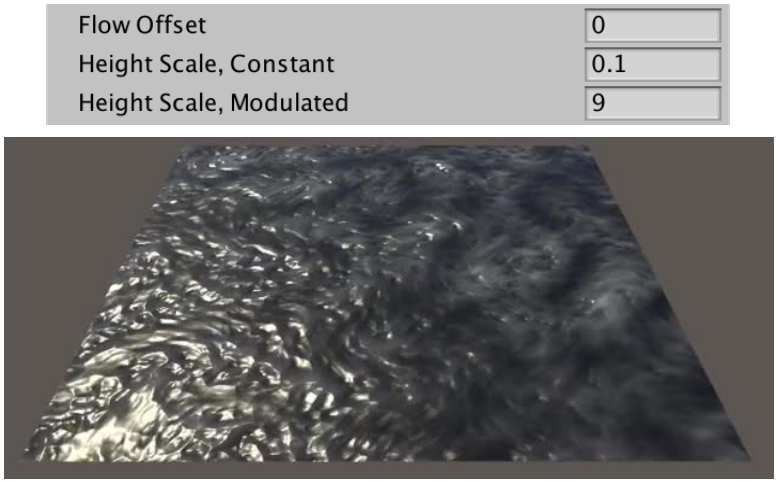
But we can go a step further. We can make the height scale variable, based on the flow speed. The idea is that you get higher waves when there is strong flow, and lower waves when there is weak flow. To control this, add a second height scale property, for the modulated height based on flow speed. The other property remains a constant scale. The final height scale is found by combining both.

```
        _HeightScale ("Height Scale, Constant", Float) = 0.25
        _HeightScaleModulated ("Height Scale, Modulated", Float) = 0.75
```

The flow speed is equal to the length of the flow vector. Multiply it by the modulating scale, then add the constant scale, and use that as the final scale for the derivatives plus height.

```
        float _HeightScale, _HeightScaleModulated;

        …

        void surf (Input IN, inout SurfaceOutputStandard o) {
            …

            float finalHeightScale =
                length(flowVector) * _HeightScaleModulated + _HeightScale;

            float3 dhA =
                UnpackDerivativeHeight(tex2D(_DerivHeightMap, uvwA.xy)) *
                (uvwA.z * finalHeightScale);
            float3 dhB =
                UnpackDerivativeHeight(tex2D(_DerivHeightMap, uvwB.xy)) *
                (uvwB.z * finalHeightScale);
            …
        }
```

While you could base the height scale purely on the flow speed, it is a good idea to use at least a small constant scale, so the surface doesn't become flat where there is no flow. For example, use a constant scale of 0.1 and a modulated scale of 9. They don't need to add up to 1, the settings depend both on how strong you want the final normals to be and how much variety you want.

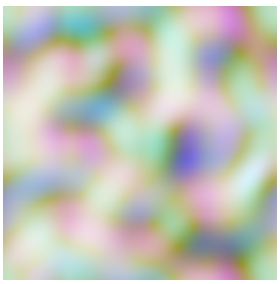


Constant plus modulated height strength.

4.5 Flow Plus Speed

Rather than calculate the flow speed in the shader, we can store it in the flow map. While filtering during sampling can change the length of vectors nonlinearly, this difference only becomes significant when two very different vectors are interpolated. That would only be the case if there were sudden directional changes in our flow map. As long as we don't have those, sampling stored speed vectors produces almost the same result. Plus, it's not essential to get an exact match when modulating the height scale.

Here is the same flow map as before, but now with the speed values stored in its B channel.



Flow map with speed in B channel.

Use the sampled data instead of calculating the speed ourselves. As speed doesn't have a direction, it should not be converted, unlike the velocity vector.

```
		void surf (Input IN, inout SurfaceOutputStandard o) {
//			float flowVector = tex2D(_FlowMap, IN.uv_MainTex).rg * 2 - 1;
			float3 flow = tex2D(_FlowMap, IN.uv_MainTex).rgb;
			flow.xy = flow.xy * 2 - 1;
			flow *= _FlowStrength;
			…

			float3 uvwA = FlowUVW(
				IN.uv_MainTex, flow.xy, jump,
				_FlowOffset, _Tiling, time, false
			);
			float3 uvwB = FlowUVW(
				IN.uv_MainTex, flow.xy, jump,
				_FlowOffset, _Tiling, time, true
			);

			float finalHeightScale =
				flow.z * _HeightScaleModulated + _HeightScale;

			…
		}
```

We wrap up by restoring the original albedo. I also change the material color to a blue tint, specifically (78, 131, 169).

```
//			o.Albedo = pow(dhA.z + dhB.z, 2);
```

Final water, with maximum jump.

The most important quality of a believable water effect is how good its animated surface normals are. Once those are good, you could can add effects like more advanced reflections, transparency, and refraction. But even without those additional features, the surface will already be interpreted as water.

The next tutorial is Directional Flow.

Enjoying the tutorials? Are they useful? Want more?

Please support me on Patreon!

Or make a direct donation!

made by Jasper Flick