Romuald Aucouturier

2D Water Shader : ici

Utilités :

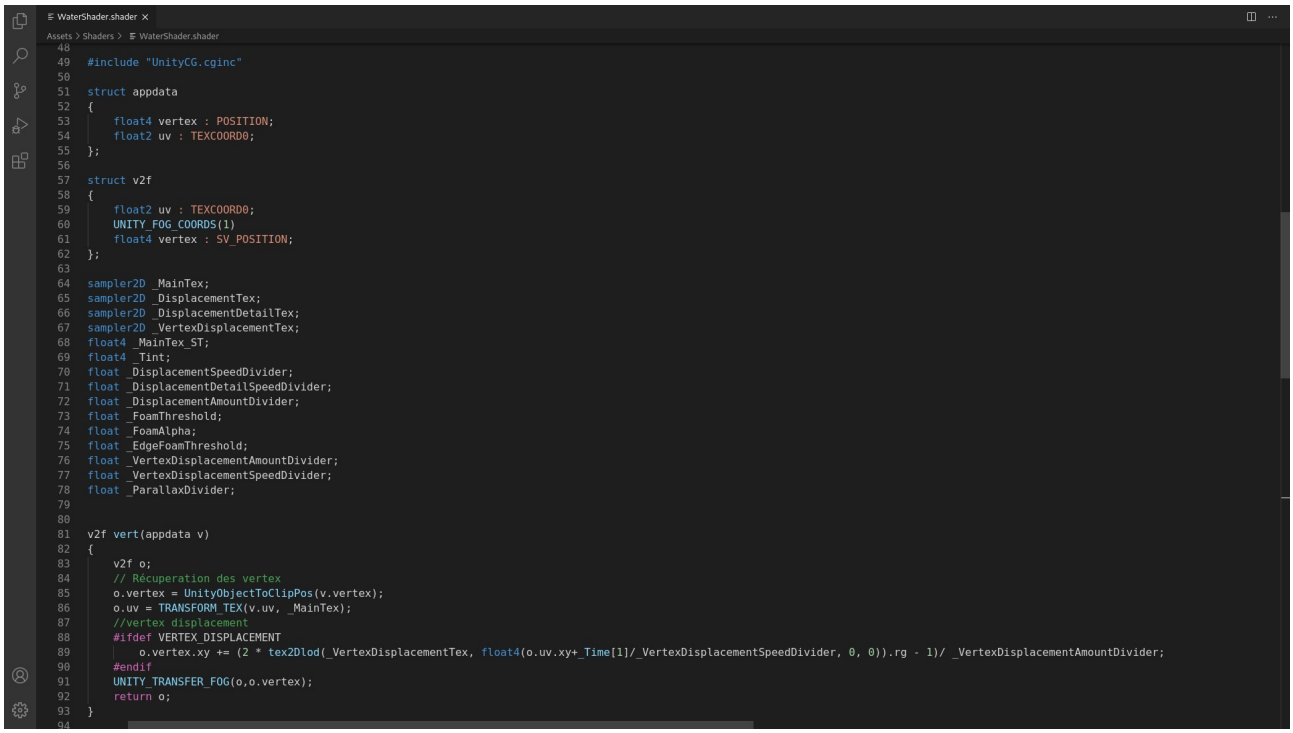
Ce shader permet de faire un rendu d'eau en 2D en fonction de la texture mise en entrée qui sera la couleur principale du rendu final.
L'utilisation d'une seconde caméra qui enverra son rendu dedans et le transmettra au shader.
Des effets sont ajoutés par défaut comme du flou, de l'ajout de nuage, et du mouvement.
D'autres effets comme le déplacement du visuel pour simuler des vagues peuvent être activés, ou de la correction visuelle.

Fonctionnement :

Le shader va prendre en entrée une texture et va l'inverser sur l'axe Y pour faire l'effet de reflet.
De la correction de perspective peut également être activée (lignes 121 - 127) pour améliorer le rendu final.
Pour l'effet des nuages un offset va permettre de savoir la taille qu'il prend sur le rendu final, et également le choix de sa transparence peut être choisi.
Pour le flou on utilise également une normal map ainsi qu'un coefficient pour récupérer la couleur correspondante.
De l'ajustement de couleur sera également utilisé pour rendre le tout un peut plus gris avec la couleur initiale (fonction *Grayscale*) de la texture et corrigera également le contraste (fonction *AdjustConstrastCurve*).

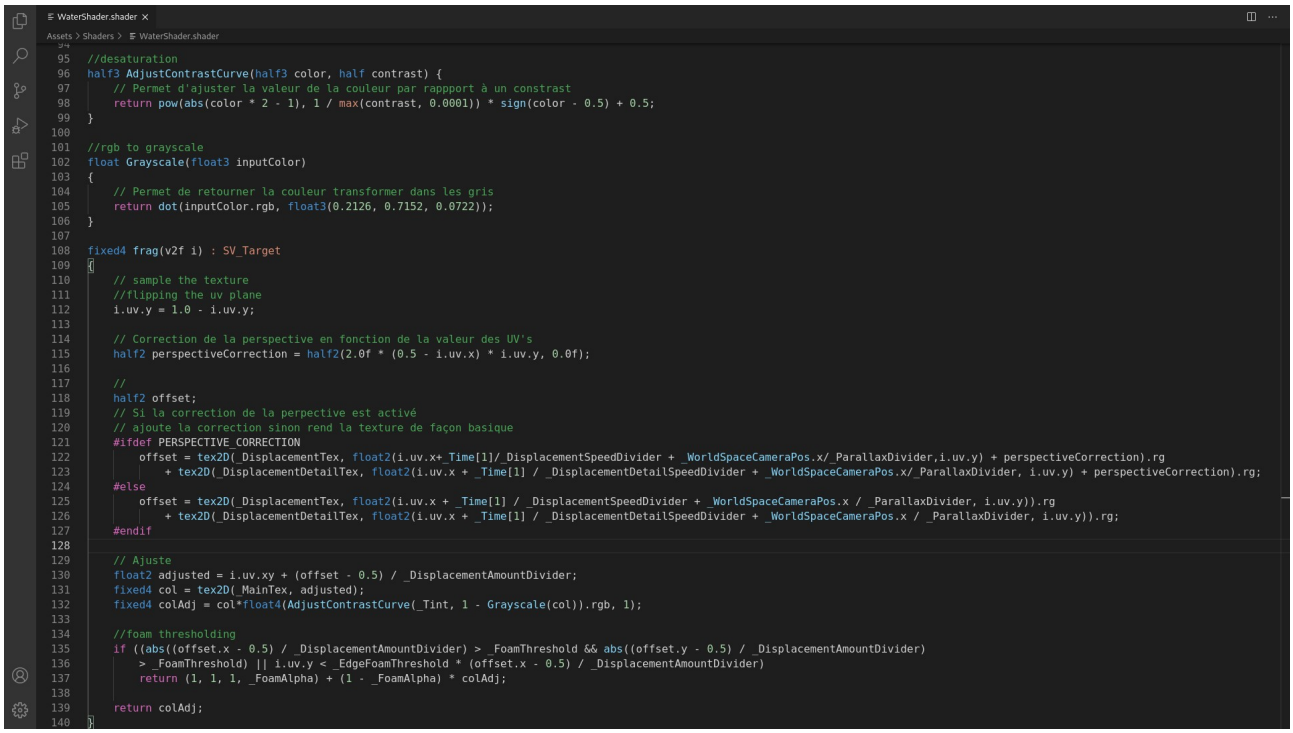
Code :

Les Paramètres et le vertex shader.



```
#include "UnityCG.cginc"

struct appdata
{
    float4 vertex : POSITION;
    float2 uv : TEXCOORD0;
};

struct v2f
{
    float2 uv : TEXCOORD0;
    UNITY_FOG_COORDS(1)
    float4 vertex : SV_POSITION;
};

sampler2D _MainTex;
sampler2D _DisplacementTex;
sampler2D _DisplacementDetailTex;
sampler2D _VertexDisplacementTex;
float4 _MainTex_ST;
float4 _Tint;
float _DisplacementSpeedDivider;
float _DisplacementDetailSpeedDivider;
float _DisplacementAmountDivider;
float _FoamThreshold;
float _FoamAlpha;
float _EdgeFoamThreshold;
float _VertexDisplacementAmountDivider;
float _VertexDisplacementSpeedDivider;
float _ParallaxDivider;


v2f vert(appdata v)
{
    v2f o;
    // Récuperation des vertex
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.uv = TRANSFORM_TEX(v.uv, _MainTex);
    //vertex displacement
    #ifdef VERTEX_DISPLACEMENT
        o.vertex.xy += (2 * tex2Dlod(_VertexDisplacementTex, float4(o.uv.xy+_Time[1]/_VertexDisplacementSpeedDivider, 0, 0)).rg - 1)/ _VertexDisplacementAmountDivider;
    #endif
    UNITY_TRANSFER_FOG(o,o.vertex);
    return o;
}
```

Le Fragment shader ainsi que les fonctions utilisés.



```
//desaturation
half3 AdjustContrastCurve(half3 color, half contrast) {
    // Permet d'ajuster la valeur de la couleur par rappport à un constrast
    return pow(abs(color * 2 - 1), 1 / max(contrast, 0.0001)) * sign(color - 0.5) + 0.5;
}

//rgb to grayscale
float Grayscale(float3 inputColor)
{
    // Permet de retourner la couleur transformer dans les gris
    return dot(inputColor.rgb, float3(0.2126, 0.7152, 0.0722));
}

fixed4 frag(v2f i) : SV_Target
{
    // sample the texture
    //flipping the uv plane
    i.uv.y = 1.0 - i.uv.y;

    // Correction de la perspective en fonction de la valeur des UV's
    half2 perspectiveCorrection = half2(2.0f * (0.5 - i.uv.x) * i.uv.y, 0.0f);

    //
    half2 offset;
    // Si la correction de la perpective est activé
    // ajoute la correction sinon rend la texture de façon basique
    #ifdef PERSPECTIVE_CORRECTION
        offset = tex2D(_DisplacementTex, float2(i.uv.x+_Time[1]/_DisplacementSpeedDivider + _WorldSpaceCameraPos.x/_ParallaxDivider,i.uv.y) + perspectiveCorrection).rg
            + tex2D(_DisplacementDetailTex, float2(i.uv.x + _Time[1] / _DisplacementDetailSpeedDivider + _WorldSpaceCameraPos.x/_ParallaxDivider, i.uv.y) + perspectiveCorrection).rg;
    #else
        offset = tex2D(_DisplacementTex, float2(i.uv.x + _Time[1] / _DisplacementSpeedDivider + _WorldSpaceCameraPos.x / _ParallaxDivider, i.uv.y)).rg
            + tex2D(_DisplacementDetailTex, float2(i.uv.x + _Time[1] / _DisplacementDetailSpeedDivider + _WorldSpaceCameraPos.x / _ParallaxDivider, i.uv.y)).rg;
    #endif

    // Ajuste
    float2 adjusted = i.uv.xy + (offset - 0.5) / _DisplacementAmountDivider;
    fixed4 col = tex2D(_MainTex, adjusted);
    fixed4 colAdj = col*float4(AdjustContrastCurve(_Tint, 1 - Grayscale(col)).rgb, 1);

    //foam thresholding
    if ((abs((offset.x - 0.5) / _DisplacementAmountDivider) > _FoamThreshold && abs((offset.y - 0.5) / _DisplacementAmountDivider)
        > _FoamThreshold) || i.uv.y < _EdgeFoamThreshold * (offset.x - 0.5) / _DisplacementAmountDivider)
        return (1, 1, 1, _FoamAlpha) + (1 - _FoamAlpha) * colAdj;

    return colAdj;
}
```

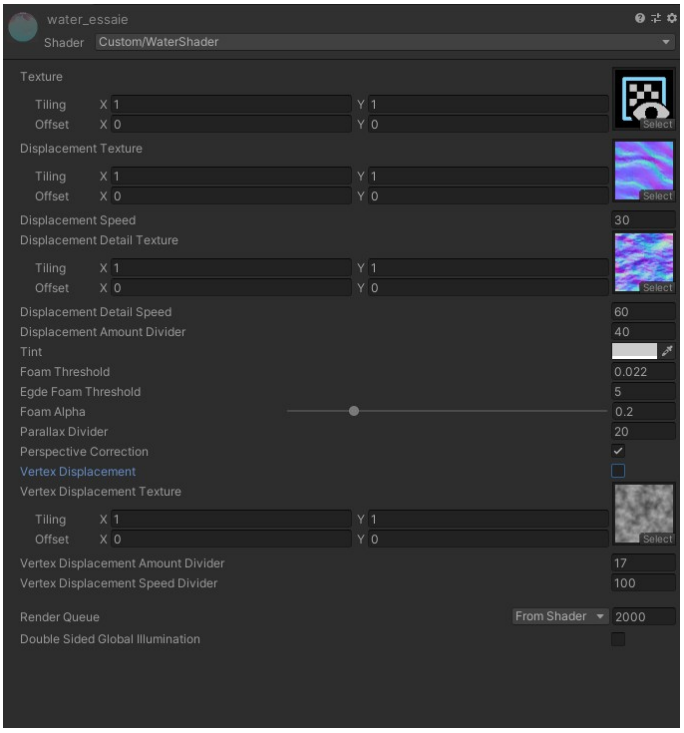
Mise en place Unity :

Pour la mise en place du shader commençai par la création d'une Render Texture dans les dossiers.
Dans la scène créer une deuxième caméra, mettre dans Target Texture la render Render Texture créé précédemment.
Ne pas oublier de mettre toute les caméras en Orthographique dans Projection, car la perspective va dégrader la qualité de la visualisation.
La caméra ayant la texture de rendu mettre le bas de la caméra en haut de la ou sera placé l'eau.
Mettre la caméra principal à l'endroit souhaité.
Créer une plane et la rendre visible dans l'univers 2D en jouant avec les rotations (90, -90, 90), la mettre de la même taille que le visuel de la caméra pour ne pas avoir de déformations.
Mettre le haut de la plane au haut de la position de l'eau.

Mettre l'environnement souhaité dans le visuel de la caméra principal, la seconde caméra dois avoir la même valeur en x que la caméra principal.

Résultats :

Fonctionnalités mise en place :



Toutes les fonctionnalités ont été mise en place, comme la correction de perspective, cocher la case du Vertex Displacement pour l'activer.
Toutes les autres fonctionnalités sont activé et ne peuvent être désactivé sauf le Foam en mettent l'alpha à 0.
Les normales et le vertex displacement ont été mise à leurs emplacement.

Focus / Explication :

Vertex Shader :

```
            v2f vert(appdata v) {
                v2f o;
                // Récuperation des vertex
                o.vertex = UnityObjectToClipPos(v.vertex);
                o.uv = TRANSFORM_TEX(v.uv, _MainTex);
                //vertex displacement
#ifdef VERTEX_DISPLACEMENT
                o.vertex.xy += (2 * tex2Dlod(_VertexDisplacementTex,
                    float4(o.uv.xy+_Time[1]/_VertexDisplacementSpeedDivider, 0, 0)).rg - 1) / _VertexDisplacementAmountDivider;
#endif
                UNITY_TRANSFER_FOG(o,o.vertex);
                return o;
            }
```

Si nous utilisons le vertex de déplacement nous allons charger une texture avec une mimap, avec les uv par rapport au temps qui est divisé par la Vitesse de déplacement.

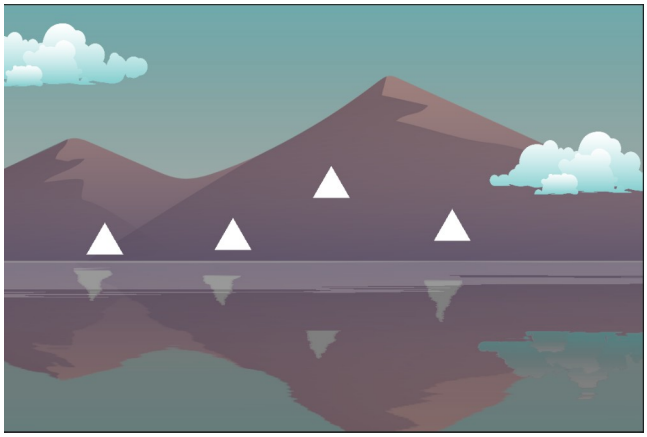
Fragment Shader :

```
// sample the texture
// flipping the uv plane
i.uv.y = 1.0 - i.uv.y;

// Variable pour le flou
half2 offset;

// Position de l'uv en x par rapport au temps
float iuvxt = i.uv.x + _Time[1];
float wp = _WorldSpaceCameraPos.x / _ParallaxDivider;
```

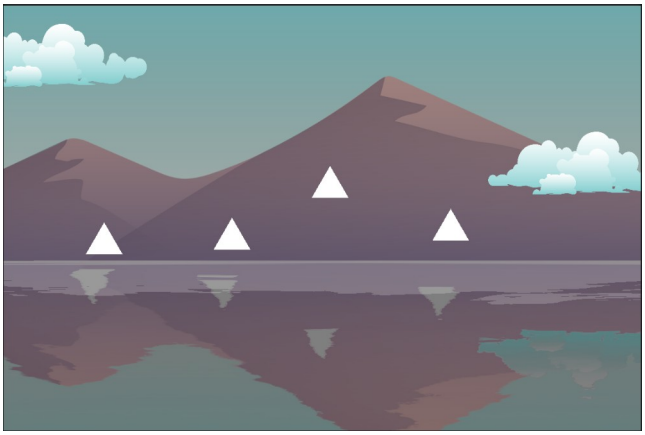
Dans le fragment shader le premier calcul est d'inverser l'uv en y pour faire l'effet de reflet car nous travaillons sur une texture donnée par Unity.

Ensuite nous calculons l'uv de x en ajoutant du temps pour ajouter un effet de mouvement.

Nous calculons la parallaxe avec la position de la camera dans le monde. La parallaxe est l'effet de changement de position de l'observateur sur sa vision. Faire ce calcul va nous permettre d'enlever la différence de cadrage avec la texture donnée ce qui nous donnera le plan comme il était vu par la caméra et non une texture donné.



Sans



Avec

Sans correction de perspective nous prenons la valeur en rouge et vert (rg) des textures de déplacement et de détail de déplacement en divisant iuvxt (l'uv en x à qui ont a ajouté du temps) par la vitesse de déplacement et de la vitesse de détail de déplacement.

```
                // récupére les informations de la texture par rapport à la position de l'uv en x
                // diviser par DisplacementDetailSpeedDivider plus la position de la caméra diviser par le parallax
#ifdef PERSPECTIVE_CORRECTION
                // Correction de la perspective en fonction de la valeur des UV's
                half2 perspectiveCorrection = half2(2 * (0.5 - i.uv.x) * i.uv.y, 0);

                // On ajoute la correction de perspective au UV
                offset = tex2D(_DisplacementTex, float2(iuvxt / _DisplacementSpeedDivider + wp, i.uv.y) + perspectiveCorrection).rg
                    + tex2D(_DisplacementDetailTex, float2(iuvxt / _DisplacementDetailSpeedDivider + wp, i.uv.y) + perspectiveCorrection).rg;
#else
                offset = tex2D(_DisplacementTex, float2(iuvxt / _DisplacementSpeedDivider + wp, i.uv.y)).rg
                    + tex2D(_DisplacementDetailTex, float2(iuvxt / _DisplacementDetailSpeedDivider + wp, i.uv.y)).rg;
#endif
```

Si la correction de perspective est activé le coefficient de correction est ajouté à l'uv calculé précédemment. Pour calculer la correction nous prenons la valeur en x de l'uv ou on soustrait 0.5 et multiplie par 2 pour avoir la valeur comprise entre -1 et 0.5. Et on multiplie par l'uv en y pour finir.

```
// Ajuste l'UV pour qu'il soit entre 0 et 1
float2 adjusted = i.uv.xy + (offset - 0.5) / _DisplacementAmountDivider;

// Get color of main texture to ajusted uv position
fixed4 col = tex2D(_MainTex, adjusted);
// Ajuste la couleur
fixed4 colAdj = col * float4(AdjustContrastCurve(_Tint, 1 - Grayscale(col)).rgb, 1);
```

Nous calculons l'ajustement pour avoir la bonne couleur de la texture à refléter.
Pour cela nous prenons les uv en x et y ou l'on ajoute l'offset calculer précédemment ou l'on enlève 0.5 et divise par le montant de déplacement.

Une fois la couleur récupérer nous l'ajustons avec la fonction ajustement de contraste en passant comme contraste le calcul de la couleur dans les gris.

```
//rgb to grayscale
float Grayscale(float3 inputColor) {
    // Permet de retourner la couleur transformer dans les gris
    return dot(inputColor.rgb, float3(0.2126, 0.7152, 0.0722));
}
```

Pour passer la couleur dans les gris, chaque couleur primaire (RGB) est multiplier par une coefficient particulier.

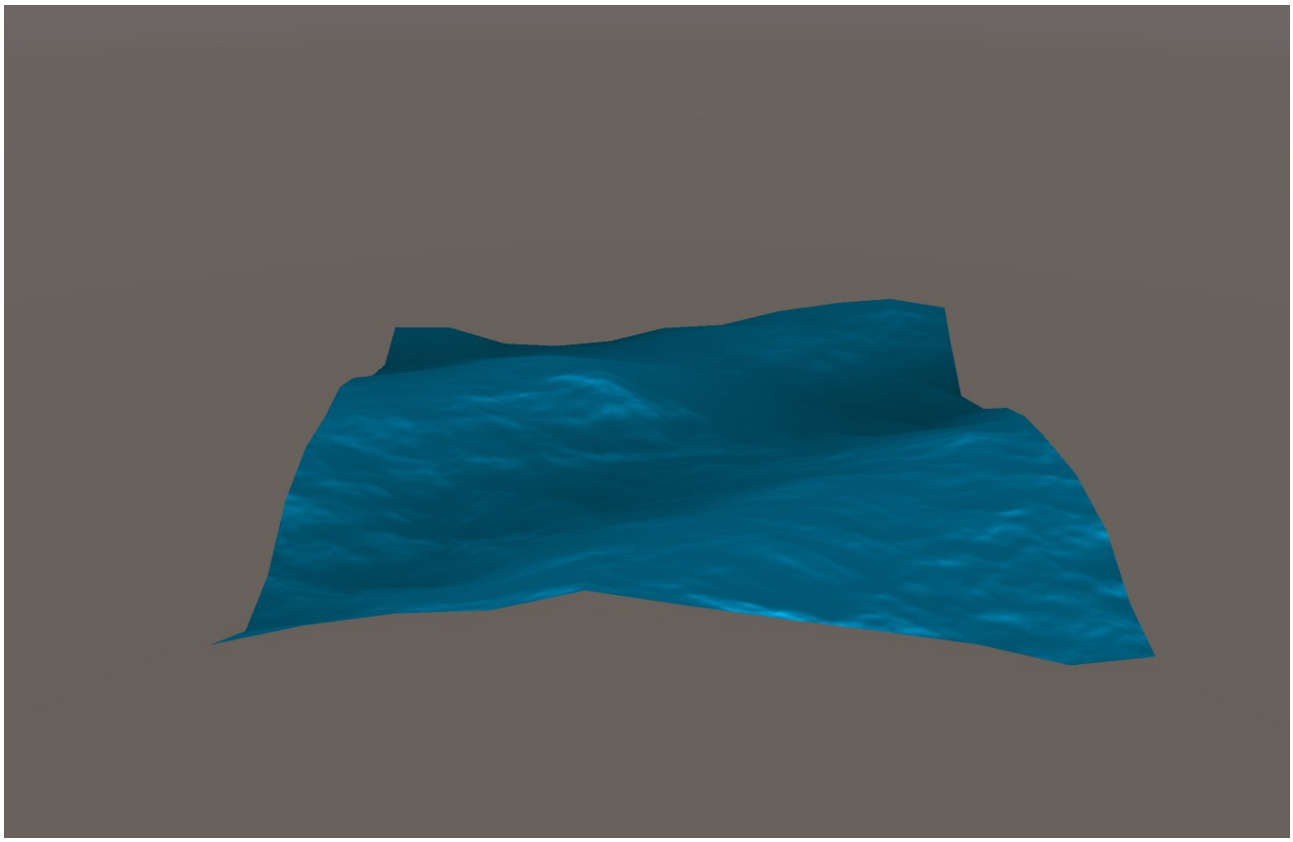
Pour ajuster le contraste nous récupérons l'absolue de la couleur pour augmenter la couleur, nous la mettons à la puissance de la valeur maximal entre le contraste donnée et 0,0001 pour ne pas réaliser une opération à 0.

```
//desaturation
half3 AdjustContrastCurve(half3 color, half contrast) {
    // Permet d'ajuster la valeur de la couleur par rappport à un constrast
    return pow(abs(color * 2 - 1), 1 / max(contrast, 0.0001)) * sign(color - 0.5) + 0.5;
}
```

La vérification sert à savoir si le shader va appliquer une surcouche blanche avec de l'alpha pour ajouter l'effet de nuage en haut de la texture.

```
//foam thresholding
if ((abs((offset.x - 0.5) / _DisplacementAmountDivider) > _FoamThreshold && abs((offset.y - 0.5) / _DisplacementAmountDivider)
    > _FoamThreshold) || i.uv.y < _EdgeFoamThreshold * (offset.x - 0.5) / _DisplacementAmountDivider)
    return (1, 1, 1, _FoamAlpha) + (1 - _FoamAlpha) * colAdj;

return colAdj;
```

3D Shader Water :



Utilités :

Ce shader permet de faire un rendu d'eau en 3D en fonction de la texture mise en entrée qui sera la couleur principale du rendu final.
L'utilisation de normal map, height map va permettre de réaliser le mouvement des vagues et en fonction de la direction du vent donnée.
D'autres paramètres permettent de choisir la couleur, la taille des vagues.

Fonctionnement :

Le shader va prendre en entrée des textures, normals map, pour la réflexion ainsi que le déplacement des vagues.
La direction des vagues est donnée par la direction du vent en vecteur 2, soit X et Y.
Pour l'effet de foam il faut mettre en entrée 2 textures ainsi que d'informations qui seront utilisées.
Ce shader possède aussi de la réfraction donnée seulette par des variables comme des floats et des vecteurs.
Il possède la possibilité de désactiver chaque élément, fonctionnalités, effets.

Code :

```
Shader "Water" {
    Properties{
        [Header(Features)]
        [Toggle(USE_DISPLACEMENT)] _UseDisplacement("Displacement", Float) = 0
        [Toggle(USE_MEAN_SKY_RADIANCE)] _UseMeanSky("Mean sky radiance", Float) = 0
        [Toggle(USE_FILTERING)] _UseFiltering("Filtering", Float) = 0
        [Toggle(USE_FOAM)] _UseFoam("Foam", Float) = 0
        [Toggle(BLINN_PHONG)] _UsePhong("Blinn Phong", Float) = 0

        [Header(Basic settings)]
        _AmbientDensity("Ambient Intensity",  Range(0, 1)) = 0.15
        _DiffuseDensity("Diffuse Intensity",  Range(0, 1)) = 0.1
        _SurfaceColor("Surface Color", Color) = (0.0078, 0.5176, 0.7)
        _ShoreColor("Shore Tint Color", Color) = (0.0078, 0.5176, 0.7)
        _DepthColor("Deep Color", Color) = (0.0039, 0.00196, 0.145)
        [NoScaleOffset]_SkyTexture("Sky Texture", Cube) = "white" {}
        [NoScaleOffset]_NormalTexture("Normal Texture", 2D) = "white" {}
        _NormalIntensity("Normal Intensity",  Range(0, 1)) = 0.5
        _TextureTiling("Texture Tiling", Float) = 1
        _WindDirection("Wind Direction", Vector) = (3,5,0)

        [Header(Displacement settings)]
        [NoScaleOffset]_HeightTexture("Height Texture", 2D) = "white" {}
        _HeightIntensity("Height Intensity",  Range(0, 1)) = 0.5
        _WaveTiling("Wave Tiling", Float) = 1
        _WaveAmplitudeFactor("Wave Amplitude Factor",Float) = 1.0
        _WaveSteepness("Wave Steepness", Range(0, 1)) = 0.5
        _WaveAmplitude("Waves Amplitude", Vector) = (0.05, 0.1, 0.2, 0.3)
        _WavesIntensity("Waves Intensity", Vector) = (3, 2, 2, 10)
        _WavesNoise("Waves Noise", Vector) = (0.05, 0.15, 0.03, 0.05)

        [Header(Refraction settings)]
        _WaterClarity("Water Clarity",  Range(0, 3)) = 0.75
        _WaterTransparency("Water Transparency",  Range(0, 30)) = 10.0
        _HorizontalExtinction("Horizontal Extinction", Vector) = (3.0, 10.0, 12.0)
        _RefractionValues("Refraction/Reflection", Vector) = (0.3, 0.01, 1.0)
        _RefractionScale("Refraction Scale",  Range(0, 0.03)) = 0.005

        [Header(Reflection settings)]
        _Shininess("Shininess",  Range(0, 3)) = 0.5
        _SpecularValues("Specular Intensity", Vector) = (12, 768, 0.15)
        _Distortion("Distortion", Range(0, 0.15)) = 0.05
        _RadianceFactor("Radiance Factor", Range(0, 1.0)) = 1.0
        [HideInInspector]_ReflectionTexture("Reflection Texture", 2D) = "white" {}

        [Header(Foam settings)]
        [NoScaleOffset]_FoamTexture("Foam Texture", 2D) = "white" {}
        [NoScaleOffset]_ShoreTexture("Shore Texture", 2D) = "white" {}
```

```
        _FoamTiling("Foam Tiling", Vector) = (2.0, 0.5, 0.0)
        _FoamRanges("Foam Ranges", Vector) = (2.0, 3.0, 100.0)
        _FoamNoise("Foam Noise", Vector) = (0.1, 0.3, 0.1, 0.3)
        _FoamSpeed("Foam Speed", Float) = 10
        _FoamIntensity("Foam Intensity", Range(0, 1)) = 0.5
        _ShoreFade("Shore Fade",  Range(0.1, 3)) = 0.3
    }
    SubShader{
        Tags{
        "IgnoreProjector" = "True"
        "Queue" = "Transparent"
        "RenderType" = "Transparent"
        }
        GrabPass{ "_RefractionTexture" }
        Pass{
        Name "Base"
        Tags{ "LightMode" = "ForwardBase" }
        Blend SrcAlpha OneMinusSrcAlpha
        Cull False
        ZWrite True

        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #include "UnityCG.cginc"
        #include "conversion.cginc"
        #include "hlsl/snoise.cginc"
        #include "hlsl/bicubic.cginc"
        #include "hlsl/normals.cginc"
        #include "hlsl/water/displacement.cginc"
        #include "hlsl/water/meansky.cginc"
        #include "hlsl/water/radiance.cginc"
        #include "hlsl/water/depth.cginc"
        #include "hlsl/water/foam.cginc"
        #pragma multi_compile_fog
        #pragma shader_feature USE_DISPLACEMENT
        #pragma shader_feature USE_MEAN_SKY_RADIANCE
        #pragma shader_feature USE_FILTERING
        #pragma shader_feature USE_FOAM
        #pragma shader_feature BLINN_PHONG
        #pragma exclude_renderers d3d11_9x
        #pragma target 3.0

        uniform sampler2D _CameraDepthTexture;
        uniform sampler2D _HeightTexture;
        uniform sampler2D _NormalTexture;
```

```
95          uniform sampler2D _FoamTexture;
96          uniform sampler2D _ShoreTexture;
97          uniform sampler2D _ReflectionTexture; uniform float4 _ReflectionTexture_TexelSize;
98          uniform samplerCUBE _SkyTexture;
99          uniform sampler2D _RefractionTexture;
100
101         uniform float4x4 _ViewProjectInverse;
102
103         uniform float4 _TimeEditor;
104         uniform float _AmbientDensity;
105         uniform float _DiffuseDensity;
106         uniform float _HeightIntensity;
107         uniform float _NormalIntensity;
108         uniform float _TextureTiling;
109
110         uniform float4 _LightColor0;
111         uniform float3 _SurfaceColor;
112         uniform float3 _ShoreColor;
113         uniform float3 _DepthColor;
114         // Wind direction in world coordinates, amplitude encoded as the length of the vector
115         uniform float2 _WindDirection;
116         uniform float _WaveTiling;
117         uniform float _WaveSteepness;
118         uniform float _WaveAmplitudeFactor;
119         // Displacement amplitude of multiple waves, x = smallest waves, w = largest waves
120         uniform float4 _WaveAmplitude;
121         // Intensity of multiple waves, affects the frequency of specific waves, x = smallest waves, w = largest waves
122         uniform float4 _WavesIntensity;
123         // Noise of multiple waves, x = smallest waves, w = largest waves
124         uniform float4 _WavesNoise;
125         // Affects how fast the colors will fade out, thus, use smaller values (eg. 0.05f).
126         // to have crystal clear water and bigger to achieve "muddy" water.
127         uniform float _WaterClarity;
128         // Water transparency along eye vector
129         uniform float _WaterTransparency;
130         // Horizontal extinction of the RGB channels, in world coordinates.
131         // Red wavelengths dissapear(get absorbed) at around 5m, followed by green(75m) and blue(300m).
132         uniform float3 _HorizontalExtinction;
133         uniform float _Shininess;
134         // xy = Specular intensity values, z = shininess exponential factor.
135         uniform float3 _SpecularValues;
136         // x = index of refraction constant, y = refraction intensity
137         // if you want to empasize reflections use values smaller than 0 for refraction intensity.
138         uniform float2 _RefractionValues;
139         // Amount of wave refraction, of zero then no refraction.
140         uniform float _RefractionScale;
```

```
141         // Reflective radiance factor.
142         uniform float _RadianceFactor;
143         // Reflection distortion, the higher the more distortion.
144         uniform float _Distortion;
145         // x = range for shore foam, y = range for near shore foam, z = threshold for wave foam
146         uniform float3 _FoamRanges;
147         // x = noise for shore, y = noise for outer
148         // z = speed of the noise for shore, y = speed of the noise for outer, not that speed can be negative
149         uniform float4 _FoamNoise;
150         uniform float2 _FoamTiling;
151         // Extra speed applied to the wind speed near the shore
152         uniform float _FoamSpeed;
153         uniform float _FoamIntensity;
154         uniform float _ShoreFade;
```

```
155
156         struct VertexInput {
157             float4 vertex : POSITION;
158         };
159
160         struct VertexOutput {
161             float4 pos : SV_POSITION;
162             float2 uv : TEXCOORD0;
163             float3 normal : TEXCOORD1;  // world normal
164             float3 tangent : TEXCOORD2;
165             float3 bitangent : TEXCOORD3;
166             float3 worldPos : TEXCOORD4;
167             float4 projPos : TEXCOORD5;
168             float timer : TEXCOORD6;
169             float4 wind : TEXCOORD7; // xy = normalized wind, zw = wind multiplied with timer
170             UNITY_FOG_COORDS(8)
171         };
```

```
173         VertexOutput vert(VertexInput v)
174         {
175             VertexOutput o = (VertexOutput)0;
176
177             float2 windDir = _WindDirection;
178             float windSpeed = length(_WindDirection);
179             windDir /= windSpeed;
180             float timer = (_Time + _TimeEditor) * windSpeed * 10;
181
182             float4 modelPos = v.vertex;
183             float3 worldPos = mul(unity_ObjectToWorld, float4(modelPos.xyz, 1));
184             half3 normal = half3(0, 1, 0);
185
186 #ifdef USE_DISPLACEMENT
187             // Récupére la distance de la camera par rapport à l'object
188             float cameraDistance = length(_WorldSpaceCameraPos.xyz - worldPos);
189             // récupére la valeur du noise par rapport à la position dans le monde.
190             // Le temps, la direction du vent
191             float2 noise = GetNoise(worldPos.xz, timer * windDir * 0.5);
192
193             half3 tangent;
194             // Les paramètres des wagues
195             float4 waveSettings = float4(windDir, _WaveSteepness, _WaveTiling);
196             // L'emplitude de la vague
197             float4 waveAmplitudes = _WaveAmplitude * _WaveAmplitudeFactor;
198             // récupére la position du monde en utilisant une displacement map.
199             // par rapport à la distance avec la caméra
200             worldPos = ComputeDisplacement(worldPos, cameraDistance, noise, timer,
201                 waveSettings, waveAmplitudes, _WavesIntensity, _WavesNoise,
202                 normal, tangent);
203
204             // add extra noise height from a heightmap
205             float heightIntensity = _HeightIntensity * (1.0 - cameraDistance / 100.0) * _WaveAmplitude;
206             // récupére les coordonnées de texture
207             float2 texCoord = worldPos.xz * 0.05 *_TextureTiling;
208
209             // Si l'intensité d'hauteur est supérieur à 0.02
210             // récupére la par rapport à la hauteur du noise
211             if (heightIntensity > 0.02) {
212                 float height = ComputeNoiseHeight(_HeightTexture, _WavesIntensity, _WavesNoise,
213                     texCoord, noise, timer);
214                 // Ajoute la taille sur la position en y dans le monde
215                 worldPos.y += height * heightIntensity;
216             }
217
```

```
217
218             // Change la position du monde avec la fonction mul de Unity
219             modelPos = UnityViewToClipPos(float4(worldPos, 1));
220             // modelPos = mul(unity_WorldToObject, float4(worldPos, 1));
221             o.tangent = tangent;
222             o.bitangent = cross(normal, tangent);
223 #endif
224             float2 uv = worldPos.xz;
225
226             o.timer = timer;
227             o.wind.xy = windDir;
228             o.wind.zw = windDir * timer;
229
230             o.uv = uv  * 0.05 * _TextureTiling;
231             o.pos = UnityObjectToClipPos(modelPos);
232             o.worldPos = worldPos;
233             o.projPos = ComputeScreenPos(o.pos);
234             o.normal = normal;
235
236             UNITY_TRANSFER_FOG(o, o.pos);
237
238             return o;
239         }
```

```
241         float4 frag(VertexOutput fs_in, float facing : VFACE) : COLOR
242         {
243             float timer = fs_in.timer;
244             float2 windDir = fs_in.wind.xy;
245             float2 timedWindDir = fs_in.wind.zw;
246             float2 ndcPos = float2(fs_in.projPos.xy / fs_in.projPos.w);
247             float3 eyeDir = normalize(_WorldSpaceCameraPos.xyz - fs_in.worldPos);
248             float3 surfacePosition = fs_in.worldPos;
249             half3 lightColor = _LightColor0.rgb;
250
251             //wave normal
252 #ifdef USE_DISPLACEMENT
253             half3 normal = ComputeNormal(_NormalTexture, surfacePosition.xz, fs_in.uv,
254                 fs_in.normal, fs_in.tangent, fs_in.bitangent, _WavesNoise, _WavesIntensity, timedWindDir);
255 #else
256             half3 normal = ComputeNormal(_NormalTexture, surfacePosition.xz, fs_in.uv,
257                 fs_in.normal, 0, 0, _WavesNoise, _WavesIntensity, timedWindDir);
258 #endif
259             normal = normalize(lerp(fs_in.normal, normalize(normal), _NormalIntensity));
260
261             // compute refracted color
262             float depth = tex2Dproj(_CameraDepthTexture, UNITY_PROJ_COORD(fs_in.projPos.xyww));
263             float3 depthPosition = NdcToWorldPos(_ViewProjectInverse, float3(ndcPos, depth));
264             float waterDepth = surfacePosition.y - depthPosition.y; // horizontal water depth
265             float viewWaterDepth = length(surfacePosition - depthPosition); // water depth from the view direction(water accumulation)
266             float2 dudv = ndcPos;
267             {
268                 // refraction based on water depth
269                 float refractionScale = _RefractionScale * min(waterDepth, 1.0f);
270                 float2 delta = float2(sin(timer + 3.0f * abs(depthPosition.y)),
271                                       sin(timer + 5.0f * abs(depthPosition.y)));
272                 dudv += windDir * delta * refractionScale;
273             }
```

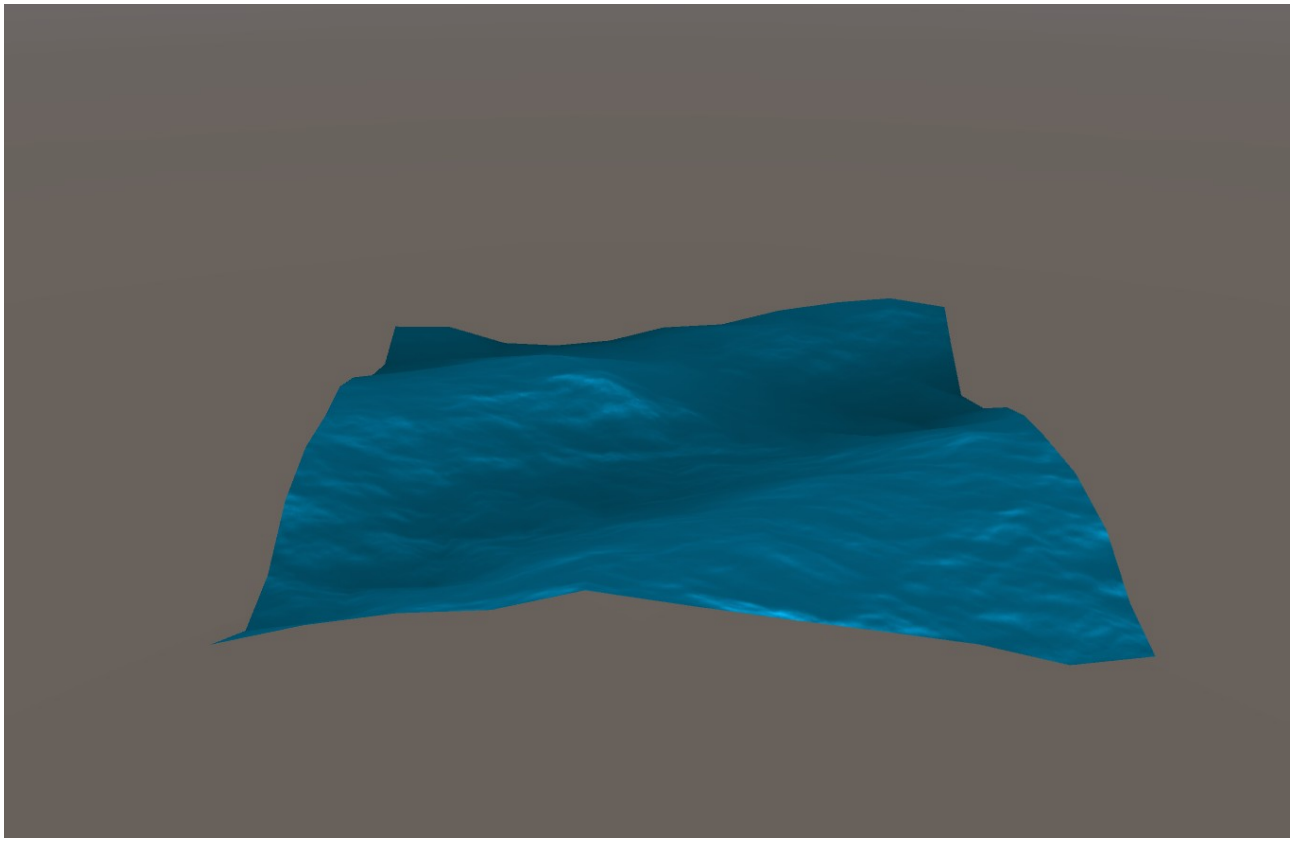
```
            half3 pureRefractionColor = tex2D(_RefractionTexture, dudv).rgb;
            {
                // reverse existing applied fog for correct shore color
                INVERSE_FOG_COLOR(fs_in.fogCoord, pureRefractionColor);
            }
            float2 waterTransparency = float2(_WaterClarity, _WaterTransparency);
            float2 waterDepthValues = float2(waterDepth, viewWaterDepth);
            float shoreRange = max(_FoamRanges.x, _FoamRanges.y) * 2.0;
            half3 refractionColor = DepthRefraction(waterTransparency, waterDepthValues, shoreRange, _HorizontalExtinction,
                                                    pureRefractionColor, _ShoreColor, _SurfaceColor, _DepthColor);

            // compute ligths's reflected radiance
            float3 lightDir = normalize(_WorldSpaceLightPos0);
            half fresnel = FresnelValue(_RefractionValues, normal, eyeDir);
            half3 specularColor = ReflectedRadiance(_Shininess, _SpecularValues, lightColor, lightDir, eyeDir, normal, fresnel);

            // compute sky's reflected radiance
#ifdef USE_MEAN_SKY_RADIANCE
            half3 reflectColor = fresnel * MeanSkyRadiance(_SkyTexture, eyeDir, normal) * _RadianceFactor;
#else
            half3 reflectColor = 0;
#endif // #ifndef USE_MEAN_SKY_RADIANCE

            // compute reflected color
            dudv = ndcPos + _Distortion * normal.xz;
#ifdef USE_FILTERING
            reflectColor += tex2DBicubic(_ReflectionTexture, _ReflectionTexture_TexelSize.z, dudv).rgb;
#else
            reflectColor += tex2D(_ReflectionTexture, dudv).rgb;
#endif // #ifdef USE_FILTERING

            // shore foam
#ifdef USE_FOAM
            float maxAmplitude = max(max(_WaveAmplitude.x, _WaveAmplitude.y), _WaveAmplitude.z);
            half foam = FoamValue(_ShoreTexture, _FoamTexture, _FoamTiling,
                _FoamNoise, _FoamSpeed * windDir, _FoamRanges, maxAmplitude,
                surfacePosition, depthPosition, eyeDir, waterDepth, timedWindDir, timer);
            foam *= _FoamIntensity;
#else
            half foam = 0;
#endif // #ifdef USE_FOAM

            half  shoreFade = saturate(waterDepth * _ShoreFade);
            // ambient + diffuse
            half3 ambientColor = UNITY_LIGHTMODEL_AMBIENT.rgb * _AmbientDensity + saturate(dot(normal, lightDir)) * _DiffuseDensity;
```

```
            // refraction color with depth based color
            pureRefractionColor = lerp(pureRefractionColor, reflectColor, fresnel * saturate(waterDepth / (_FoamRanges.x * 0.4)));
            pureRefractionColor = lerp(pureRefractionColor, _ShoreColor, 0.30 * shoreFade);
            // compute final color
            half3 color = lerp(refractionColor, reflectColor, fresnel);
            color = saturate(ambientColor + color + max(specularColor, foam * lightColor));
            color = lerp(pureRefractionColor + specularColor * shoreFade, color, shoreFade);
            UNITY_APPLY_FOG(fs_in.fogCoord, color);

#ifdef DEBUG_NORMALS
            color.rgb = 0.5 + 2 * ambientColor + specularColor + clamp(dot(normal, lightDir), 0, 1) * 0.5;
#endif

            return float4(color * _SurfaceColor, 1.0);
        }
        ENDCG
        }
    }
    CustomEditor "WaterShaderGUI"
    FallBack "Diffuse"
}
```

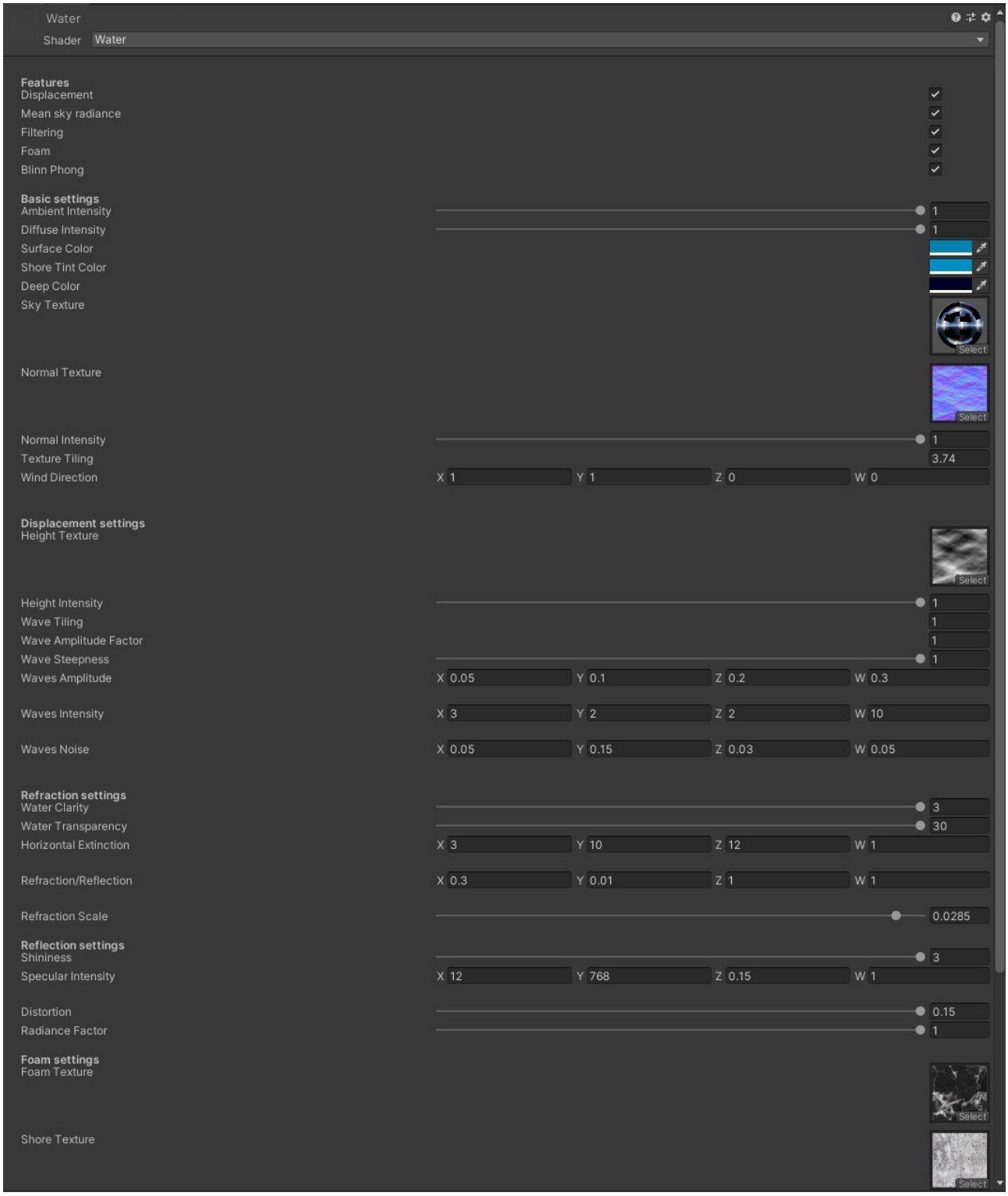
Mise en place Unity :

Pour la mise en place du shader commençai par la création d’une plane dans la scène.
Mettre la plane visible par la caméra.
Créer la texture par rapport au shader, et placer la texture sur la plane.
Vous devriez déjà voir des modifications sur le visuel de la plane.
Ensuite mettre les textures, Normal Map, Height Map, dans le shader pour avoir un rendu plus réaliste.
Dans la partie feature activer les effets visuels souhaités.

Résultats :



Fonctionnalités mise en place :



Toutes les fonctionnalités et effets ont été mis en place comme le déplacement des vertex pour les vagues, aux reflets de Phong et les autres effets, éléments.
Les textures, Normal Map ont été mise à leur place.

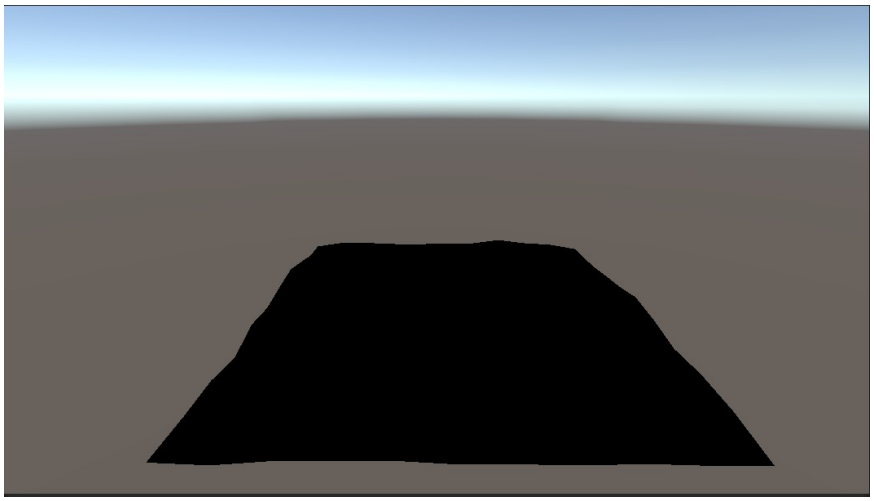
Focus / Explication :

L'explication va seulement se porter sur les vagues, et le fonctionnement dans le vertex et fragment shader.

Vertex Shader :

```
187             // Récupére la distance de la camera par rapport à l'object
188             float cameraDistance = length(_WorldSpaceCameraPos.xyz - worldPos);
189             // récupére la valeur du noise par rapport à la position dans le monde.
190             // Le temps, la direction du vent
191             float2 noise = GetNoise(worldPos.xz, timer * windDir * 0.5);
192
193             half3 tangent;
194             // Les paramètres des wagues
195             float4 waveSettings = float4(windDir, _WaveSteepness, _WaveTiling);
196             // L'emplitude de la vague
197             float4 waveAmplitudes = _WaveAmplitude * _WaveAmplitudeFactor;
198             // récupére la position du monde en utilisant une displacement map.
199             // par rapport à la distance avec la caméra
200             worldPos = ComputeDisplacement(worldPos, cameraDistance, noise, timer,
201                 waveSettings, waveAmplitudes, _WavesIntensity, _WavesNoise,
202                 normal, tangent);
```

Cette partie va récupérer les éléments essentiels pour pouvoir lancer la fonction *ComputeDisplacement* qui va permettre d'avoir la position du vertex dans le monde. En fonction de la position de la caméra, d'un effet de noise, et de la configuration des vagues.



Voici le résultat si on enlève l'appel de la fonction *ComputeDisplacement*.

Nous pouvons voir que l'élément devient noir et non bleu et que le déplacement est moins important et qu'il ne se fait pas en fonction de la direction du vent.

ComputeDisplacement :

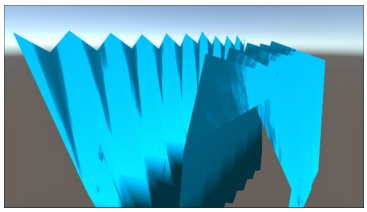
```
77      float2 windDir = waveSettings.xy;
78      float waveSteepness = waveSettings.z;
79      float waveTiling = waveSettings.w;
```

Cette partie va seulement servir à récupérer les paramètres des vagues donnés.

```
84      wavesIntensity = normalize(wavesIntensity);
85      waveNoise = half4(noise.x - noise.x * 0.2 + noise.y * 0.1, noise.x + noise.y * 0.5 - noise.y * 0.1, noise.x, noise.x) * waveNoise;
86      half4 wavelengths = half4(1, 4, 3, 6) + waveNoise;
87      half4 amplitudes = waveAmplitudes + half4(0.5, 1, 4, 1.5) * waveNoise;
```

Ce bloc de code va permettre d'arrondir les vagues et dons de les rendre moins brutes.
Voici ce qui se passe sans cette partie.

Nous pouvons voir que les vagues sont beaucoup trop hautes et ne reflète pas comment se comporte une vague.

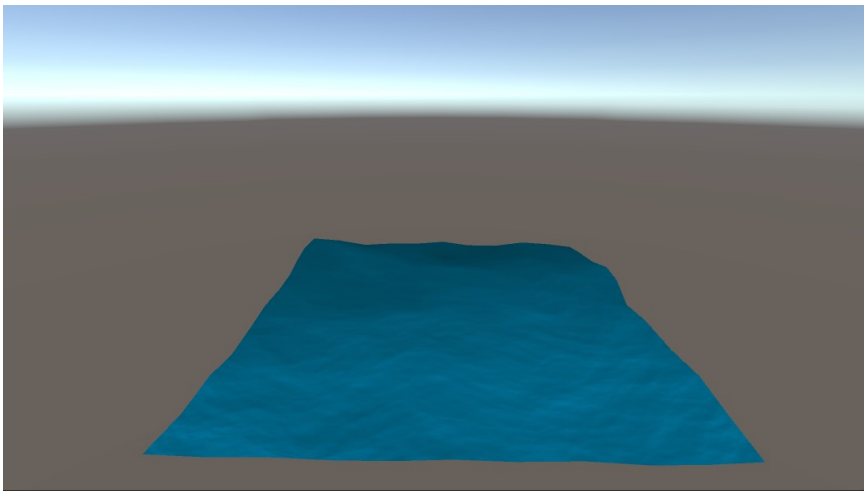


```
89      // reduce wave intensity base on distance to reduce aliasing
90      wavesIntensity *= 1.0 - saturate(half4(cameraDistance / 120.0, cameraDistance / 150.0, cameraDistance / 170.0, cameraDistance / 400.0));
```

Comme écrit en commentaire cette partie va permettre de réduire l'intensité des vagues en fonction de la distance pour réduire l'aliasing. Aucune différence visuelle est vraiment constatée si cette partie est enlevée.

```
92      // compute position and normal from several sine and gerstner waves
93      tangent = normal = half3(0, 1, 0);
94      float2 timers = float2(timer * 0.5, timer * 0.25);
95      for (int i = 2; i < 4; ++i) {
96          float A = wavesIntensity[i] * amplitudes[i];
97          float3 vals = SineWaveValues(worldPos.xz * waveTiling, windDir, A, wavelengths[i], timers);
98          normal += wavesIntensity[i] * SineWaveNormal(windDir, A, vals);
99          tangent += wavesIntensity[i] * SineWaveTangent(windDir, A, vals);
100         worldPos.y += SineWaveDelta(A, vals);
101     }
```

Cette partie va permettre de données l'aspect aux vagues donc de faire le changement de position dans le monde en Y.
Il va calculer la valeur de la vague en du vent, de la position dans le monde, du temps et de la force de la vague qui servira à calculer la normal, la tangente et son delta c'est-à-dire sa position dans le monde.
Par la suite la normal et la tangente seront calculés en fonction du vent, de sa valeur et d'A qui est son amplitude.

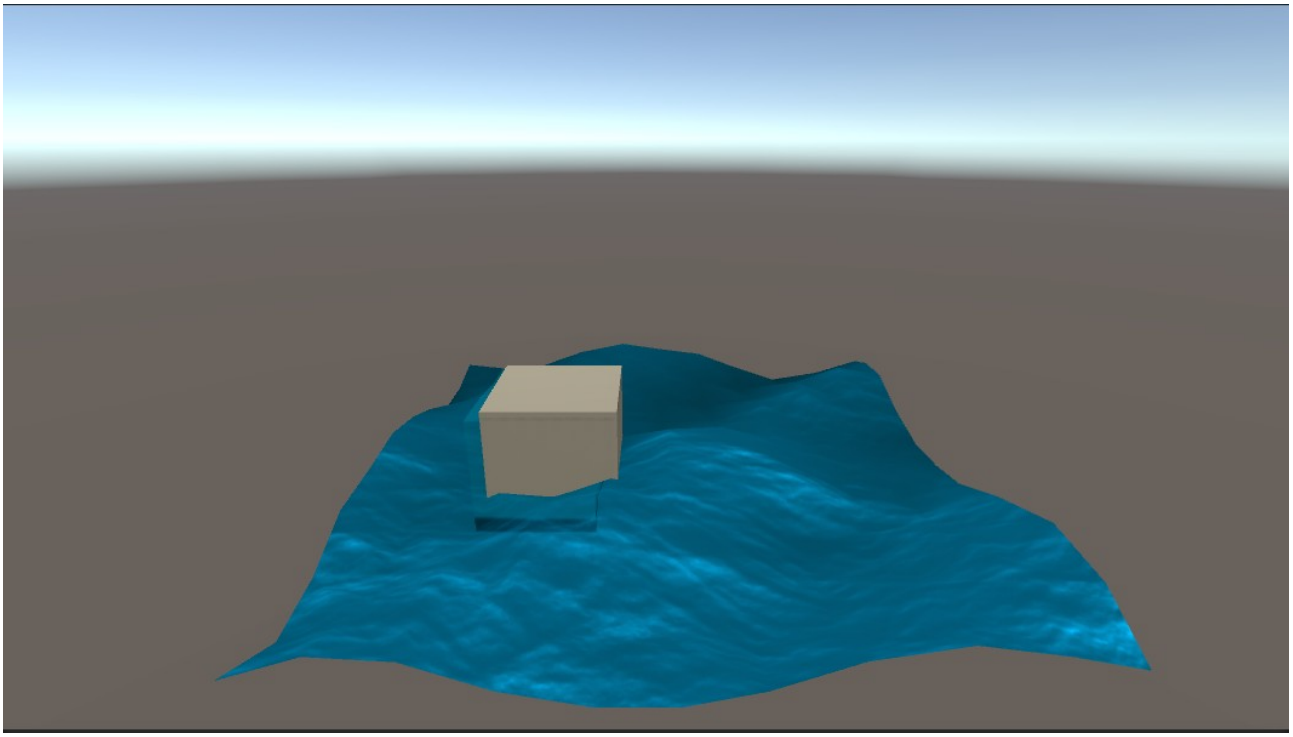


Nous pouvons voir que les vagues ne sont pas calculés sans cette partie, et que le reflet qui utilise la normal et la height map n'a donc pas le même rendu vu que ces valeurs ne sont pas calculées.

```
103     // using normalized wave steepness, tranform to Q
104     float2 Q = waveSteepness / ((2 * 3.14159265 / wavelengths.xy) * amplitudes.xy);
105     for (int j = 0; j < 2; ++j) {
106         float A = wavesIntensity[j] * amplitudes[j];
107         float3 vals = GerstnerWaveValues(worldPos.xz * waveTiling, windDir, A, wavelengths[j], Q[j], timer);
108         normal += wavesIntensity[j] * GerstnerWaveNormal(windDir, A, Q[j], vals);
109         tangent += wavesIntensity[j] * GerstnerWaveTangent(windDir, A, Q[j], vals);
110         worldPos += GerstnerWaveDelta(windDir, A, Q[j], vals);
111     }
```

On utilise la normal de la vague pour calculer le Gerstner ou la Houle trochoïdale.
Qui est la description des ondes de gravités de forme périodique qui se propagent à la surface d'un fluide. C'est ce qui va donc permettre de calculer quand un obstacle se trouve dans l'eau.

Le fonctionnement global est quasiment identique à celui précédent car il recalcule sa valeur, sa normal, sa tangente ainsi que son delta.

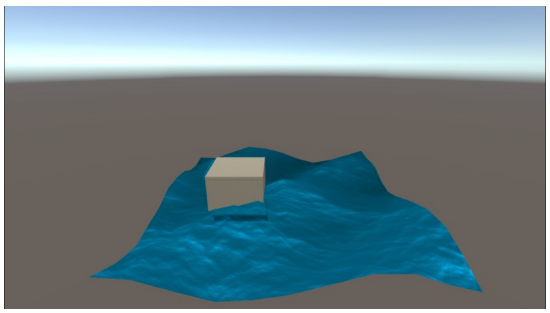


Voici ce qu'il se passe avec un objet, mais que ce soit avec cette partie de code active ou non le visuel ne change pas assez pour le remarquer.

```
113     normal = normalize(normal);
114     tangent = normalize(tangent);
115     if (length(wavesIntensity) < 0.01) {
116         normal = half3(0, 1, 0);
117         tangent = half3(0, 0, 1);
118     }
119
120     return worldPos;
```

Dans cette fin de fonction on recalcule la normal et la tangente en les normalisant et vérifie que si l'intensité de la vague est trop faible met une valeur pas défaut.
Et la fonction finie en retournant la valeur de la position dans le monde.

Le résultat ne change pas assez même si on désactive cette partie de la fonction.



Vertex Shader :

```
204             // add extra noise height from a heightmap
205             float heightIntensity = _HeightIntensity * (1.0 - cameraDistance / 100.0) * _WaveAmplitude;
206             // récupére les coordonnées de texture
207             float2 texCoord = worldPos.xz * 0.05 *_TextureTiling;
208 
209             // Si l'intensité d'hauteur est supérieur à 0.02
210             // récupére la par rapport à la hauteur du noise
211             if (heightIntensity > 0.02) {
212                 float height = ComputeNoiseHeight(_HeightTexture, _WavesIntensity, _WavesNoise,
213                     texCoord, noise, timer);
214                 // Ajoute la taille sur la position en y dans le monde
215                 worldPos.y += height * heightIntensity;
216             }
```

On commence par ajuster l'intensité de la hauteur en fonction de l'amplitude de la vague, et de la distance de la caméra.

Si l'intensité est au-dessus de 0,02 on ajoute donc à la position Y dans le monde une hauteur multiplier par son intensité.

ComputeNoiseHeight :

```
56  float ComputeNoiseHeight(sampler2D heightTexture, float4 wavesIntensity, float4 wavesNoise, float2 texCoord, float2 noise, float2 timedWindDir)
57  {
58      AdjustWavesValues(noise, wavesNoise, wavesIntensity);
59  
60      float2 texCoords[4] = { texCoord * 1.6 + timedWindDir * 0.064 + wavesNoise.x,
61                              texCoord * 0.8 + timedWindDir * 0.032 + wavesNoise.y,
62                              texCoord * 0.5 + timedWindDir * 0.016 + wavesNoise.z,
63                              texCoord * 0.3 + timedWindDir * 0.008 + wavesNoise.w };
64      float height = 0;
65      for (int i = 0; i < 4; ++i) {
66          height += tex2Dlod(heightTexture, float4(texCoords[i], 0, 0)).x * wavesIntensity[i];
67      }
68  
69      return height;
70  }
```

On commence par ajuster la valeur de la vague avec *AdjustWavesValues* et on va s'en servir pour créer un tableau de coordonnée pour itérer 4 fois la récupération de la valeur height dans la texture donnée (*heightTexture*).

Et retourne la valeur height calculé.

Vertex Shader :

```
218             // Change la position du monde avec la fonction mul de Unity
219             modelPos = UnityViewToClipPos(float4(worldPos, 1));
220             // modelPos = mul(unity_WorldToObject, float4(worldPos, 1));
221             o.tangent = tangent;
222             o.bitangent = cross(normal, tangent);
```

Ce sont les fonctions basiques utilisées par Unity pour calculer la position dans le monde de l'élément et applique la tangente et la bitangente.

Fragment Shader :

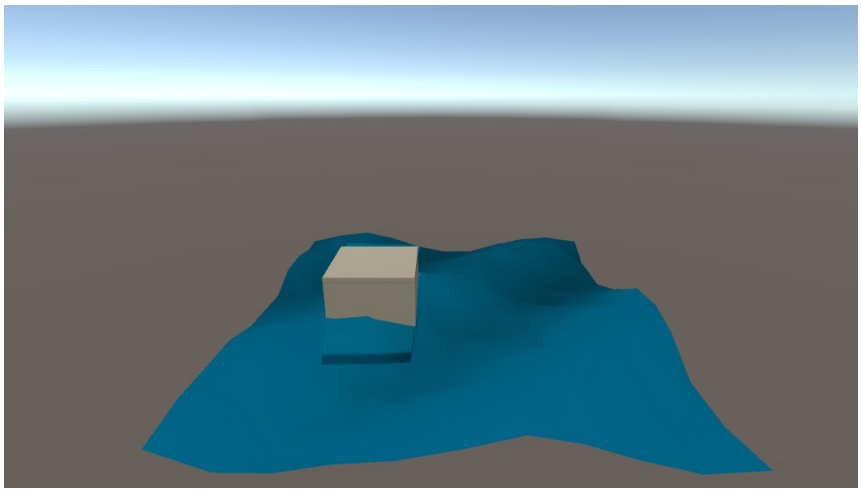
```
252  #ifdef USE_DISPLACEMENT
253              half3 normal = ComputeNormal(_NormalTexture, surfacePosition.xz, fs_in.uv,
254                  fs_in.normal, fs_in.tangent, fs_in.bitangent, _WavesNoise, _WavesIntensity, timedWindDir);
255  #else
256              half3 normal = ComputeNormal(_NormalTexture, surfacePosition.xz, fs_in.uv,
257                  fs_in.normal, 0, 0, _WavesNoise, _WavesIntensity, timedWindDir);
258  #endif
```

Si on calcule les vagues nous devons donc calculer la normal en fonction des tangentes.

ComputeNormal :

```
22  // uv in texture space, normal in world space
23  half3 ComputeNormal(sampler2D normalTexture, float2 worldPos, float2 texCoord,
24      half3 normal, half3 tangent, half3 bitangent,
25      half4 wavesNoise, half4 wavesIntensity, float2 timedWindDir)
26  {
27      float2 noise = GetNoise(worldPos, timedWindDir * 0.5);
28      AdjustWavesValues(noise, wavesNoise, wavesIntensity);
29
30      float2 texCoords[4] = { texCoord * 1.6 + timedWindDir * 0.064 + wavesNoise.x,
31          texCoord * 0.8 + timedWindDir * 0.032 + wavesNoise.y,
32          texCoord * 0.5 + timedWindDir * 0.016 + wavesNoise.z,
33          texCoord * 0.3 + timedWindDir * 0.008 + wavesNoise.w };
34
35      half3 wavesNormal = half3(0, 1, 0);
36  #ifdef USE_DISPLACEMENT
37      normal = normalize(normal);
38      tangent = normalize(tangent);
39      bitangent = normalize(bitangent);
40      for (int i = 0; i < 4; ++i)
41      {
42          wavesNormal += ComputeSurfaceNormal(normal, tangent, bitangent, normalTexture, texCoords[i]) * wavesIntensity[i];
43      }
44  #else
45      for (int i = 0; i < 4; ++i)
46      {
47          wavesNormal += UnpackNormal(tex2D(normalTexture, texCoords[i])) * wavesIntensity[i];
48      }
49      wavesNormal.xyz = wavesNormal.xzy; // flip zy to avoid btn multiplication
50  #endif // #ifdef USE_DISPLACEMENT
51
52      return wavesNormal;
53  }
```

Le début se passe comme dans la fonction *ComputeNoiseHeight* mais c'est seulement dans le calcul de la normal que cela diffère en fonction de si on calcule les vagues ou non.



Dans le calcul on n'utilise non pas juste une texture mais on calcule la compute surface normal en fonction de la normal et des tangentes. Ce qui permet d'avoir le reflet de la lumière et non pas comme le montre la capture sans la partie qui est sans aucune réflexion de lumière sur l'eau.

Et c'est ce qui donne les effets de vagues dans ce shader.