Listing 1: Darstellung des Wasserbassins (Vertex Shader)

```
#version 330
precision highp float;

#define ATTR_POSITION  0
[...]
#define ATTR_TEXCOORD7 9

layout(location = ATTR_POSITION)  in vec4 gs_Vertex;
[...]
layout(location = ATTR_TEXCOORD7) in vec4 gs_MultiTexCoord7;

out vec4 gs_TexCoord[8];

uniform mat4 matWorldViewProjection;

void main()
{
    gl_Position    = matWorldViewProjection * gs_Vertex;
    gs_TexCoord[0] = vec4(gs_Vertex.xyz, gl_Position.z);
}
```

Listing 2: Darstellung des Wasserbassins (Fragment Shader)

```
#version 330

precision highp float;

in  vec4 gs_TexCoord[8];
out vec4 gs_FragColor[5];

void main()
{
    gs_FragColor[0] = vec4(0.0, 0.0, 0.0, 1.0);

    // SceneCameraSpacePosAndDepth:
    gs_FragColor[1] = gs_TexCoord[0];

    gs_FragColor[2] = vec4(0.0, 0.0, 0.0, 1.0);
    gs_FragColor[3] = vec4(0.0, 0.0, 0.0, 1.0);
    gs_FragColor[4] = vec4(0.0, 0.0, 0.0, 1.0);
}
Ende
```

Listing 3: Eigenschaften einer Wasserfläche zwischenspeichern (Vertex Shader)

```
#version 330
precision highp float;

#define ATTR_POSITION  0
[...]
#define ATTR_TEXCOORD7 9

layout(location = ATTR_POSITION)  in vec4 gs_Vertex;
[...]
layout(location = ATTR_TEXCOORD7) in vec4 gs_MultiTexCoord7;
out vec4 gs_TexCoord[8];

uniform mat4 matWorldViewProjection;
uniform mat4 matCamera;

void main()
{
    gl_Position = matWorldViewProjection * gs_Vertex;
    vec4 Position = matCamera * gs_Vertex;
    gs_TexCoord[0] = vec4(Position.xyz, gl_Position.z);
}
```

Listing 4: Eigenschaften einer Wasserfläche zwischenspeichern (Fragment Shader)

```
#version 330
precision highp float;

in  vec4 gs_TexCoord[8];
out vec4 gs_FragColor;

void main()
{
    // SceneCameraSpacePosAndDepth:
    gs_FragColor = gs_TexCoord[0];
}
```

Listing 5: Deferred Water Rendering – Aufbau des Fragment Shaders

```
void main()
{
    vec4 SceneCameraSpacePosAndDepth =
        max
        (
            texture(SceneCameraSpacePosAndDepthTexture, gs_TexCoord[0].st),
            texture
            (
                SceneCameraSpacePosAndDepthTexture,
                gs_TexCoord[0].st +
                vec2(0.001, 0.001)
            )
        );

    vec4 WaterCameraSpacePosAndDepth = texture(WaterCameraSpacePosAndDepthTexture, gs_TexCoord[0].st);

    // Sicherstellen, dass Hintergrundbilder (Sky-Sphären, Billboards, etc.)
    // wie weit entfernte Hintergrundobjekte behandelt werden können!
    if(SceneCameraSpacePosAndDepth.w < 0.0)
        SceneCameraSpacePosAndDepth.w = 10000000000000.0;

    vec3 WaterCameraSpacePos = WaterCameraSpacePosAndDepth.xyz;
    vec3 NormalizedWaterCameraSpacePos = normalize(WaterCameraSpacePos);

    vec3 SceneWorldSpacePos = SceneCameraSpacePosAndDepth.xyz + CameraPosition;

    vec2 texCoord;

    [Berechnung des 3D-Welleneffekts, Listing 6]

    // Kamera sowie Szenenpixel über dem Wasser (Fall 1):
    if(CameraPosition.y > correctedWaterSurfaceHeight && WaterDepth < 0.0)
    {
        gs_FragColor = texture(ScreenTexture, gs_TexCoord[0].st);
    }

    else
    {
        [Berechnung der Wellen-Normalenvektoren, Listing 7]

        // Kamera im Wasser:
        if(CameraPosition.y < correctedWaterSurfaceHeight)
        {
            // Szenenpixel befindet sich im Wasser (Fall 2):
            if(WaterDepth > 0.0)
            {
                [Lichtabsorption des Wassers simulieren, Listing 10]
                [Unterwasser-Kaustiken berechnen, Listing 14]
                vec4 ScreenColor = texture(ScreenTexture, gs_TexCoord[0].st);
                CausticsIntensity *= (max(ScreenColor.x, max(ScreenColor.y, ScreenColor.z)));

                // Wasserfarbe ohne Wellenbewegung!!
                float diffuseIntensityWater = 0.35;

                // Neuberechnung der Pixelfarbe:
                gs_FragColor =
                    vec4(DepthColorValues * depthDependedLightIntensity, 1.0) *
                    (
                        CausticsColor *
                        CausticsIntensity +
                        ScreenColor
                    ) +
                    WaterColor *
                    vec4(diffuseIntensityWater * InvDepthColorValues, 1.0);
            }
            // Kamera im Wasser, Szenenpixel über dem Wasser (Fall 3):
            else
            {
                WaterDepth = WaterSurfaceHeight-CameraPosition.y;
                float diffuseIntensityWater = 0.35;

                [Lichtabsorption des Wassers simulieren, Listing 11]
                [Refraktions/Verzerrungs-Effekt simulieren, Listing 15]
                [spiegelnde Reflexionen auf der Wasseroberfläche, Listing 9]
                [Neuberechnung der Pixelfarbe (Kamera im Wasser, Szenenpixel oberhalb) Listing 17]
            }
        }
        // Kamera über dem Wasser, Wassertiefe größer null:
        else if(WaterDepth > 0.0)
        {
            [Berechnung der lokalen Wasserspiegelungen (Screen Space Reflections), Listing 8]
            [spiegelnde Reflexionen auf der Wasseroberfläche, Listing 9]
            [Schaumkronen u. Brandung, Listing 13]

            float InsideWaterViewDistance = SceneCameraSpacePosAndDepth.w – WaterCameraSpacePosAndDepth.w;

            float depthValueBlue =
```

```
                min
                (
                    1.0,
                    max
                    (
                        0.0,
                        (
                            InsideWaterMaxViewDistance –
                            InsideWaterViewDistance
                        ) *
                        InsideWaterInvMaxViewDistance
                    )
                );

            // Szenenpixel im Wasser in der Nähe der Kamera (Fall 4):
            if(depthValueBlue > 0.0 && SceneCameraSpacePosAndDepth.w < 2000.0)
            {
                [Lichtabsorption des Wassers simulieren, Listing 12]
                [Refraktions/Verzerrungs-Effekt simulieren, Listing 16]
                [Unterwasser-Kaustiken berechnen, Listing 14]
                [Neuberechnung der Pixelfarbe (Kamera über dem Wasser, Unterwasserpixel nahe der Kamera), Listing 18]
            }
            // Szenenpixel im Wasser fernab der Kamera (Fall 5):
            else
            {
                // Nebel- bzw. Dunst-Berechnungen:
                float OneMinusHazeValue =
                    max
                    (
                        0.0,
                        (
                            2000.0 -
                            WaterCameraSpacePosAndDepth.w
                        ) *
                        0.0005
                    );
                float HazeValue = 1.0 - OneMinusHazeValue;

                gs_FragColor =
                    HazeValue *
                    texture(BackgroundScreenTexture, gs_TexCoord[0].st) +
                    OneMinusHazeValue *
                    (
                        FoamColor +
                        diffuseIntensity *
                        (
                            WaterColor +
                            ScreenSpaceReflectionColor +
                            SpecularIntensity *
                            vec4(LightColor.xyz, 1.0) +
                            EnvironmentSpecularIntensity *
                            EnvironmentLightColor
                        )
                    );
            }
        }
        // Szenengeometrie außerhalb des Wassers (Fall 1):
        else
        {
            gs_FragColor = texture(ScreenTexture, gs_TexCoord[0].st);
        }
    }
}
```

Listing 6: Berechnung des 3D-Welleneffekts

```
float MaxCalculationStep;

if(CameraPosition.y > WaterSurfaceHeight)
{
    MaxCalculationStep = 10.0 * MaxWaveAmplitude;
}
else
{
    MaxCalculationStep = -10.0 * MaxWaveAmplitude;
}

float ActualCalculationStep = 0.5 * MaxCalculationStep;

vec3 TestCameraSpacePos =
    WaterCameraSpacePos-
    ActualCalculationStep * NormalizedWaterCameraSpacePos;

float testHeight = 0.0;
vec3 TestWorldSpacePos;
vec2 texCoordWaveSim;
```

```
float PhaseAngle, cosPhaseAngle;

for(int i = 0; i < 10; i++)
{
    TestWorldSpacePos = TestCameraSpacePos + CameraPosition;

    texCoordWaveSim.x =
        WaveValues.x * TestWorldSpacePos.x +
        0.5;
    texCoordWaveSim.y =
        0.5 -
        WaveValues.x * TestWorldSpacePos.z;

    /* einfache Sinuswelle:
    testHeight = NegWindVector.y*(sin(WaveFrontValues.w+
    NegWindVector.x*TestWorldSpacePos.x+NegWindVector.z*TestWorldSpacePos.z) +
    WaveFrontValues.x*(2.0*texture(WaterHeightTexture, texCoordWaveSim).x-1.0));*/

    // zykloid (trochoid) ähnliche Welle:
    PhaseAngle =
        WaveFrontValues.w +
        NegWindVector.x * TestWorldSpacePos.x +
        NegWindVector.z * TestWorldSpacePos.z;

    cosPhaseAngle = cos(PhaseAngle);

    testHeight =
        NegWindVector.y *
        (
            sin
            (
                PhaseAngle -
                0.5 * cosPhaseAngle
            ) +
            WaveFrontValues.x *
            (
                2.0 * texture(WaterHeightTexture, texCoordWaveSim).x -
                1.0
            )
        );

    // Hinweise:
    // NegWindVector.y und WaveFrontValues.x entsprechen den Wellenamplituden
    // WaveFrontValues.w entspricht der Simulationszeit

    ActualCalculationStep *= 0.5;

    if((TestCameraSpacePos.y - WaterCameraSpacePos.y) < testHeight)
    {
        TestCameraSpacePos =
            TestCameraSpacePos –
            ActualCalculationStep * NormalizedWaterCameraSpacePos;
    }
    else
    {
        TestCameraSpacePos =
            TestCameraSpacePos +
            ActualCalculationStep * NormalizedWaterCameraSpacePos;
    }
}

WaterCameraSpacePos = TestCameraSpacePos;
vec3 WaterWorldSpacePos = WaterCameraSpacePos + CameraPosition;
float correctedWaterSurfaceHeight = WaterSurfaceHeight + testHeight;
float WaterDepth = correctedWaterSurfaceHeight - SceneWorldSpacePos.y;
```

Listing 7: Berechnung der Normalenvektoren einer Wasserfläche

```
vec3 SurfaceNormal;
float diffuseIntensity;
float Height1, Height2, Height3, Height4;

// Für ein Unterwasserpixel muss kein Wellen-Normalenvektor berechnet werden,
// sofern sich die Kamera ebenfalls Unterwasser befindet:
if(!(CameraPosition.y < correctedWaterSurfaceHeight && WaterDepth > 0.0))
{
    // Zunächst einmal sorgen wir für ein wenig zusätzliche Wellenbewegung:

    float AdditionalWaveHeight =
        WaveFrontValues.x *
        (
            2.0 *
            texture(WaterHeightTexture, texCoordWaveSim).x –
            1.0
        );
```

```
Height1 =
    2.0 *
    texture
    (
        WaterHeightTexture,
        texCoordWaveSim + vec2(-WaterSurfaceNormalCalculationParameter.x, 0.0)
    ).x -
    1.0;
Height2 =
    2.0 *
    texture
    (
        WaterHeightTexture,
        texCoordWaveSim + vec2(WaterSurfaceNormalCalculationParameter.x, 0.0)
    ).x -
    1.0;
Height3 =
    2.0 *
    texture
    (
        WaterHeightTexture,
        texCoordWaveSim + vec2(0.0, -WaterSurfaceNormalCalculationParameter.x)
    ).x -
    1.0;
Height4 =
    2.0 *
    texture
    (
        WaterHeightTexture,
        texCoordWaveSim + vec2(0.0,  WaterSurfaceNormalCalculationParameter.x)
    ).x -
    1.0;

if(WaterCameraSpacePosAndDepth.w < 500.0)
{
    texCoordWaveSim *= 2.0;
    Height1 +=
        0.75 *
        (
            2.0 *
            texture
            (
                WaterHeightTexture,
                texCoordWaveSim + vec2(-WaterSurfaceNormalCalculationParameter.x, 0.0)
            ).x -
            1.0
        );
    Height2 +=
        0.75 *
        (
            2.0 *
            texture
            (
                WaterHeightTexture,
                texCoordWaveSim + vec2(WaterSurfaceNormalCalculationParameter.x, 0.0)
            ).x -
            1.0
        );
    Height3 +=
        0.75 *
        (
            2.0 *
            texture
            (
                WaterHeightTexture,
                texCoordWaveSim + vec2(0.0, -WaterSurfaceNormalCalculationParameter.x)
            ).x -
            1.0
        );
    Height4 +=
        0.75 *
        (
            2.0 *
            texture
            (
                WaterHeightTexture,
                texCoordWaveSim + vec2(0.0, WaterSurfaceNormalCalculationParameter.x)
            ).x -
            1.0
        );
}

texCoordWaveSim.x =
    WaveValues.z *
    WaterWorldSpacePos.x +
    0.5;
texCoordWaveSim.y =
    0.5 -
```

```
    WaveValues.z *
    WaterWorldSpacePos.z;

float phaseAngle =
    WaveValues.w +
    texCoordWaveSim.x *
    NegWindVector.x –
    texCoordWaveSim.y *
    NegWindVector.z;

float tempFloat = sin(phaseAngle);
Height2 *= tempFloat;
Height4 *= tempFloat;

tempFloat = cos(phaseAngle);
Height1 *= tempFloat;
Height3 *= tempFloat;

/* Mithilfe des diffuseIntensity-Parameters simulieren wir die
   Helligkeitsschwankungen der Wasserfläche.
   Die zugrundeliegende Idee ist denkbar einfach – Wellenberge
   erscheinen heller als Wellentäler, da sie mehr Licht in Richtung
   des Spielers reflektieren können:*/

float diffuseIntensity =
    WaterSurfaceNormalCalculationParameter.z *
    (Height1 + Height2 + Height3 + Height4);

vec2 heightSamplePos = vec2(TestWorldSpacePos.x + 2.1, TestWorldSpacePos.z);
float WaveHeight1 =
    WaveFrontValues.y *
    (
        AdditionalWaveHeight +
        WaveFrontValues.z *
        sin
        (
            WaveFrontValues.w +
            NegWindVector.x *
            heightSamplePos.x +
            NegWindVector.z *
            heightSamplePos.y
        )
    );

heightSamplePos =
    vec2
    (
        TestWorldSpacePos.x - 2.1,
        TestWorldSpacePos.z
    );
float WaveHeight2 =
    WaveFrontValues.y *
    (
        AdditionalWaveHeight +
        WaveFrontValues.z *
        sin
        (
            WaveFrontValues.w +
            NegWindVector.x * heightSamplePos.x +
            NegWindVector.z * heightSamplePos.y
        )
    );

heightSamplePos =
    vec2
    (
        TestWorldSpacePos.x,
        TestWorldSpacePos.z + 2.1
    );
float WaveHeight3 =
    WaveFrontValues.y *
    (
        AdditionalWaveHeight +
        WaveFrontValues.z *
        sin
        (
            WaveFrontValues.w +
            NegWindVector.x * heightSamplePos.x +
            NegWindVector.z * heightSamplePos.y
        )
    );

heightSamplePos =
    vec2
    (
        TestWorldSpacePos.x,
        TestWorldSpacePos.z - 2.1
    );
float WaveHeight4 =
```

```
        WaveFrontValues.y *
        (
            AdditionalWaveHeight +
            WaveFrontValues.z *
            sin
            (
                WaveFrontValues.w +
                NegWindVector.x * heightSamplePos.x +
                NegWindVector.z * heightSamplePos.y
            )
        );

    Height1 += WaveHeight1;
    Height2 += WaveHeight2;
    Height3 += WaveHeight3;
    Height4 += WaveHeight4;

    diffuseIntensity +=
        NegWindVector.w *
        (WaveHeight1 + WaveHeight2 + WaveHeight3 + WaveHeight4);

    diffuseIntensity = 0.35 + diffuseIntensity;

    // Berechnung des Wellen-Normalenvektors:
    SurfaceNormal =
        normalize
        (
            vec3
            (
                Height1 - Height2,
                (
                    0.5 +
                    0.025 *
                    WaterCameraSpacePosAndDepth.w
                ) *
                WaterSurfaceNormalCalculationParameter.w,
                Height3-Height4
            )
        );
}
```

Listing 8: Berechnung der lokalen Wasserspiegelungen (Screen Space Reflections)

```
vec4 ScreenSpaceReflectionColor = vec4(0.0, 0.0, 0.0, 0.0);

if(WaterCameraSpacePosAndDepth.w < ScreenSpaceReflectionRange)
{
    vec3 ReflectionSurfaceNormal =
        normalize
        (
            vec3
            (
                Height1 - Height2,
                (
                    5.0 +
                    0.025 *
                    WaterCameraSpacePosAndDepth.w
                ) *
                WaterSurfaceNormalCalculationParameter.w,
                Height3-Height4
            )
        );

    // gespiegelten Blickrichtungsvektor berechnen:
    vec3 ReflectionVector;
    vec3 ReflectionDirection = reflect(NormalizedWaterCameraSpacePos, ReflectionSurfaceNormal);

    float tempDot = dot(ReflectionDirection, ReflectionSurfaceNormal);

    float distanceStep = 7.0 / max(0.6, tempDot * tempDot);
    float actualDistanceStep = 0.5 * distanceStep;

    vec4 Projection;
    float InvW;
    float ReflectionvectorTexY;
    float ReflectionvectorTexX;
    vec3 diffVector;
    vec4 TestSceneCameraSpacePosAndDepth;
    float DistancSq;
    float distanceTestFactor = 1.0;
    int i;

    for(i = 0; i < 10; i++)
    {
        // neue Ray-Marching-Position berechnen:
        ReflectionVector =
            WaterCameraSpacePos +
            actualDistanceStep *
```

```
        ReflectionDirection;

    actualDistanceStep += distanceStep;

    Projection =
        matViewProjection *
        vec4(ReflectionVector, 0.0);

    InvW = 1.0 / Projection.w;
    ReflectionvectorTexY =
        0.5 *
        Projection.y *
        InvW +
        0.5;
    ReflectionvectorTexX =
        0.5 *
        Projection.x *
        InvW +
        0.5;

    TestSceneCameraSpacePosAndDepth =
        texture
        (
            SceneCameraSpacePosAndDepthTexture,
            vec2(ReflectionvectorTexX, ReflectionvectorTexY)
        );

    // quadratischen Abstand zwischen der aktuellen Ray-Marching-Position
    // und der Szenengeometrie ermitteln:
    diffVector = TestSceneCameraSpacePosAndDepth.xyz - ReflectionVector;
    DistancSq = dot(diffVector, diffVector);

    // Schnittpunkt gefunden?
    if(DistancSq < 16.0*distanceTestFactor)
    {
        break;
    }

    // Genauigkeit des Schnittpunkttests nach jedem
    // Testdurchlauf ein wenig vergrößern:
    distanceTestFactor *= 1.1;
}

// Falls kein Schnittpunkt gefunden wurde, spiegeln wir
// stattdessen einfach den Szenenhintergrund:
if(i == 10)
{
    Projection =
        matViewProjection *
        vec4(ReflectionDirection, 0.0);
    InvW = 1.0 / Projection.w;
    ReflectionvectorTexY =
        0.5 *
        Projection.y *
        InvW +
        0.5;
    ReflectionvectorTexX =
        0.5 *
        Projection.x *
        InvW +
        0.5;
}

// Intensität der Spiegelung in Abhängigkeit von den
// gefundenen Texturkoordinaten berechnen:
float DTexX = 2.0 * ReflectionvectorTexX - 1.0;
float DTexY = 2.0 * ReflectionvectorTexY - 1.0;

float intensity =
    1.0 /
    inversesqrt
    (
        inversesqrt
        (
            1.0 /
            (
                0.01 +
                DTexX * DTexX +
                DTexY * DTexY
            )
        )
    );
//float intensity = sqrt(sqrt(0.01+DTexX*DTexX + DTexY*DTexY));

intensity =
    max
    (
        1.0 - intensity,
        0.0
```

```
        ) *
        min
        (
            1.0,
            ScreenSpaceReflectionIntensity / WaterCameraSpacePosAndDepth.w
        );

    ScreenSpaceReflectionColor =
        intensity *
        texture
        (
            ScreenTexture,
            vec2(ReflectionvectorTexX, ReflectionvectorTexY)
        );
}
```

Listing 9: spiegelnde Reflexionen auf der Wasseroberfläche (nach Phong)

```
// spiegelnde Reflexion des Umgebungslichts:
float EnvironmentSpecularIntensity =
    max
    (
        -dot
        (
            2.0 *
                dot(SurfaceNormal, EnvironmentNegLightDir) *
                SurfaceNormal -
                EnvironmentNegLightDir,
            NormalizedWaterCameraSpacePos
        ),
        0.0
    );

// spiegelnde Reflexion des Sonnenlichts:
float SpecularIntensity =
    max
    (
        -dot
        (
            2.0 *
                dot(SurfaceNormal, NegLightDir) *
                SurfaceNormal –
                NegLightDir,
            NormalizedWaterCameraSpacePos
        ),
        0.0
    );
SpecularIntensity = pow(SpecularIntensity, LightColor.w);
```

Listing 10: Lichtabsorption des Wassers (Kamera u. Szenenpixel im Wasser)

```
float depthValueBlue =
    min
    (
        1.0,
        max
        (
            0.0,
            (
                InsideWaterMaxViewDistance –
                SceneCameraSpacePosAndDepth.w
            ) *
            InsideWaterInvMaxViewDistance
        )
    );

float depthValueGreen =
    min
    (
        1.0,
        max
        (
            0.0,
            depthValueBlue *
            (
                1.0 –
                SceneCameraSpacePosAndDepth.w *
                RelativeAbsorbtionCoeffGreen
            )
        )
    );

float depthValueRed =
    min
```

```
    (
        1.0,
        max
        (
            0.0,
            depthValueBlue *
            (
                1.0 –
                SceneCameraSpacePosAndDepth.w *
                RelativeAbsorbtionCoeffRed
            )
        )
    );

float depthDependedLightIntensity =
    min
    (
        1.0,
        InvDepthDependedLightIntensityDecrease /
        (WaterSurfaceHeight - CameraPosition.y)
    );

vec3 DepthColorValues =
    vec3
    (
        depthValueRed * depthValueRed,
        depthValueGreen * depthValueGreen,
        depthValueBlue * depthValueBlue
    );

vec3 InvDepthColorValues =
(
    vec3(1.0) -
    DepthColorValues
) *
depthDependedLightIntensity;
```

Listing 11: Lichtabsorption des Wassers (Kamera im Wasser, Szenenpixel oberhalb)

```
float depthValueBlue =
    min
    (
        1.0,
        max
        (
            0.0,
            (
                InsideWaterMaxViewDistance –
                WaterCameraSpacePosAndDepth.w
            ) *
            InsideWaterInvMaxViewDistance
        )
    );

float depthValueGreen =
    min
    (
        1.0,
        max
        (
            0.0,
            depthValueBlue *
            (
                1.0 –
                WaterCameraSpacePosAndDepth.w *
                RelativeAbsorbtionCoeffGreen
            )
        )
    );

float depthValueRed =
    min
    (
        1.0,
        max
        (
            0.0,
            depthValueBlue *
            (
                1.0 –
                WaterCameraSpacePosAndDepth.w *
                RelativeAbsorbtionCoeffRed
            )
        )
    );
```

```
float depthDependedLightIntensity =
    min
    (
        1.0,
        InvDepthDependedLightIntensityDecrease / (WaterSurfaceHeight - CameraPosition.y)
    );

vec3 DepthColorValues =
    vec3
    (
        depthValueRed * depthValueRed,
        depthValueGreen * depthValueGreen,
        depthValueBlue * depthValueBlue
    );

vec3 InvDepthColorValues =
(
    vec3(1.0) -
    DepthColorValues
) *
depthDependedLightIntensity;

// Helligkeitsabnahme mit zunehmender Wassertiefe:
DepthColorValues *= depthDependedLightIntensity;
```

Listing 12: Lichtabsorption des Wassers (Kamera über dem Wasser)

```
float depthValueBlue =
    min
    (
        1.0,
        max
        (
            0.0,
            (
                InsideWaterMaxViewDistance –
                InsideWaterViewDistance
            ) *
            InsideWaterInvMaxViewDistance
        )
    );

float depthValueGreen =
    min
    (
        1.0,
        max
        (
            0.0,
            depthValueBlue *
            (
                1.0 –
                InsideWaterViewDistance *
                RelativeAbsorbtionCoeffGreen
            )
        )
    );

float depthValueRed =
    min
    (
        1.0,
        max
        (
            0.0,
            depthValueBlue *
            (
                1.0 –
                InsideWaterViewDistance *
                RelativeAbsorbtionCoeffRed
            )
        )
    );

vec3 DepthColorValues =
    vec3
    (
        depthValueRed * depthValueRed,
        depthValueGreen * depthValueGreen,
        depthValueBlue * depthValueBlue
    );

vec3 InvDepthColorValues = vec3(1.0) - DepthColorValues;
```

Listing 13: Schaumkronen u. Brandung (Kamera über dem Wasser)

```
// Texturkoordinaten für die Schaumkronen-Darstellung:
vec2 texCoordFoam;
texCoordFoam.x =
    WaveValues.y *
    WaterWorldSpacePos.x +
    0.5;
texCoordFoam.y =
    0.5 –
    WaveValues.y *
    WaterWorldSpacePos.z;
texCoordFoam +=
    WaterSurfaceNormalCalculationParameter.y *
    vec2(SurfaceNormal.x, SurfaceNormal.z);

vec4 FoamColor =
    max
    (
        max
        (
            0.0,
            CoastalFoamIntensity *
            (
                1.0 –
                WaterDepth /
                MaxDepthForCoastalFoam
            )
        ),
        WaveFoamIntensity *
        max
        (
            0.0,
            (
                diffuseIntensity –
                WaveFoamHeightValue
            )
        )
    ) *
    min
    (
        1.0,
        4.0 / WaterCameraSpacePosAndDepth.w
    ) *
    EnvironmentLightColor *
    texture(FoamTexture, texCoordFoam);

// Intensität der spiegelnden Reflexion abschwächen
// (die Schaumkronen vermindern die Lichtreflexion!)

float FoamBasedIntensityDecrease = max(0.0,(1.0-FoamColor.x));
// Hinweis: Schaumkronen sind normalerweise weiß - daher ist es egal,
// welchen Farbkanal wir nutzen!

FoamBasedIntensityDecrease *= FoamBasedIntensityDecrease*
FoamBasedIntensityDecrease;
SpecularIntensity *= FoamBasedIntensityDecrease;
```

Listing 14: Unterwasser-Kaustiken

```
vec2 texCoordCaustics;
texCoordCaustics.x =
    CausticsValues.x *
    SceneWorldSpacePos.x +
    0.5;
texCoordCaustics.y =
    0.5 –
    CausticsValues.x *
    SceneWorldSpacePos.z;

vec2 CausticsMovementTexCoord =
    vec2
    (
        CausticsValues.w * NegWindVector.x,
        -CausticsValues.w * NegWindVector.z
    );

float CausticsIntensity =
    texture
    (
        WaterHeightTexture,
        TexCoordCaustics + CausticsMovementTexCoord
    ).x +
    texture
    (
```

```
        WaterHeightTexture,
        1.5 * texCoordCaustics – CausticsMovementTexCoord
    ).x –
    1.0;

CausticsIntensity =
    Clamp
    (
        1.0 - abs(CausticsIntensity),
        0.0,
        1.0
    );
CausticsIntensity *= CausticsIntensity * CausticsIntensity;
CausticsIntensity *= CausticsIntensity * CausticsIntensity;
CausticsIntensity *= CausticsValues.y;
```

Listing 15: Refraktions-Effekt (Kamera im Wasser)

```
// Verzerrungen der Szenengeometrie oberhalb des Wasserspiegels mithilfe
// von modifizierten Texturkoordinaten (ModifiedTexCoord) simulieren:
float modificationValue =
    1.0 -
    max
    (
        0.0,
        (
            InsideWaterMaxViewDistance –
            SceneCameraSpacePosAndDepth.w +
            WaterCameraSpacePosAndDepth.w
        ) *
        InsideWaterInvMaxViewDistance
    );

vec2 ModifiedTexCoord =
    gs_TexCoord[0].st +
    modificationValue *
    WaterSurfaceNormalCalculationParameter.y *
    vec2(SurfaceNormal.x, SurfaceNormal.z);
```

Listing 16: Refraktions-Effekt (Kamera über dem Wasser)

```
// Verzerrungen der Unterwasser-Szenengeometrie mithilfe
// von modifizierten Texturkoordinaten (ModifiedTexCoord) simulieren:
vec2 ModifiedTexCoord =
    gs_TexCoord[0].st +
    InvDepthColorValues.b *
    WaterSurfaceNormalCalculationParameter.y *
    vec2(SurfaceNormal.x, SurfaceNormal.z);
```

Listing 17: Pixelfarbe (Kamera im Wasser, Szenenpixel oberhalb)

```
// Texturkoordinaten für die Schaumkronen-Darstellung:
vec2 texCoordFoam;
texCoordFoam.x =
    WaveValues.y *
    WaterWorldSpacePos.x +
    0.5;
texCoordFoam.y =
    0.5 –
    WaveValues.y *
    WaterWorldSpacePos.z;
texCoordFoam +=
    WaterSurfaceNormalCalculationParameter.y *
    vec2(SurfaceNormal.x, SurfaceNormal.z);

gs_FragColor =
    max
    (
        max
        (
            0.0,
            0.5 * (1.0 – WaterDepth / MaxDepthForCoastalFoam)
        ),
        testHeight *
        min(1.0, 50.0 / WaterCameraSpacePosAndDepth.w)
    ) *
    EnvironmentLightColor *
    vec4(DepthColorValues, 1.0) *
    texture(FoamTexture, texCoordFoam) +
    vec4(DepthColorValues, 1.0) *
    (
```

```
        diffuseIntensity *
        SpecularIntensity *
        vec4(LightColor.xyz, 1.0) +
        DiffuseIntensity *
        EnvironmentSpecularIntensity *
        EnvironmentLightColor
    ) +
    vec4(DepthColorValues, 1.0) *
    texture(ScreenTexture, ModifiedTexCoord) +
    WaterColor *
    vec4(diffuseIntensityWater * InvDepthColorValues, 1.0);
```

Listing 18: Pixelfarbe (Kamera über dem Wasser, Unterwasserpixel nahe der Kamera)

```
vec4 ScreenColor = texture(ScreenTexture, ModifiedTexCoord);
CausticsIntensity *=
    FoamBasedIntensityDecrease *
    (
        max
        (
            ScreenColor.x,
            max(ScreenColor.y, ScreenColor.z)
        )
    );
ScreenColor *= FoamBasedIntensityDecrease;
float OneMinusHazeValue =
    max
    (
        0.0,
        (2000.0-WaterCameraSpacePosAndDepth.w) * 0.0005
    );
float HazeValue = 1.0 - OneMinusHazeValue;
gs_FragColor =
    HazeValue * texture(BackgroundScreenTexture, gs_TexCoord[0].st) +
    OneMinusHazeValue *
    (
        FoamColor + vec4(DepthColorValues, 1.0) *
        (
            CausticsColor * CausticsIntensity+ScreenColor) +
            DiffuseIntensity * (vec4(InvDepthColorValues, 1.0) * WaterColor +
            ScreenSpaceReflectionColor + SpecularIntensity * vec4(LightColor.xyz, 1.0) +
            EnvironmentSpecularIntensity * EnvironmentLightColor
        )
    );
```