Physically Correct perturbed planar Refractions&Reflections in Real Time - Shader Documentation

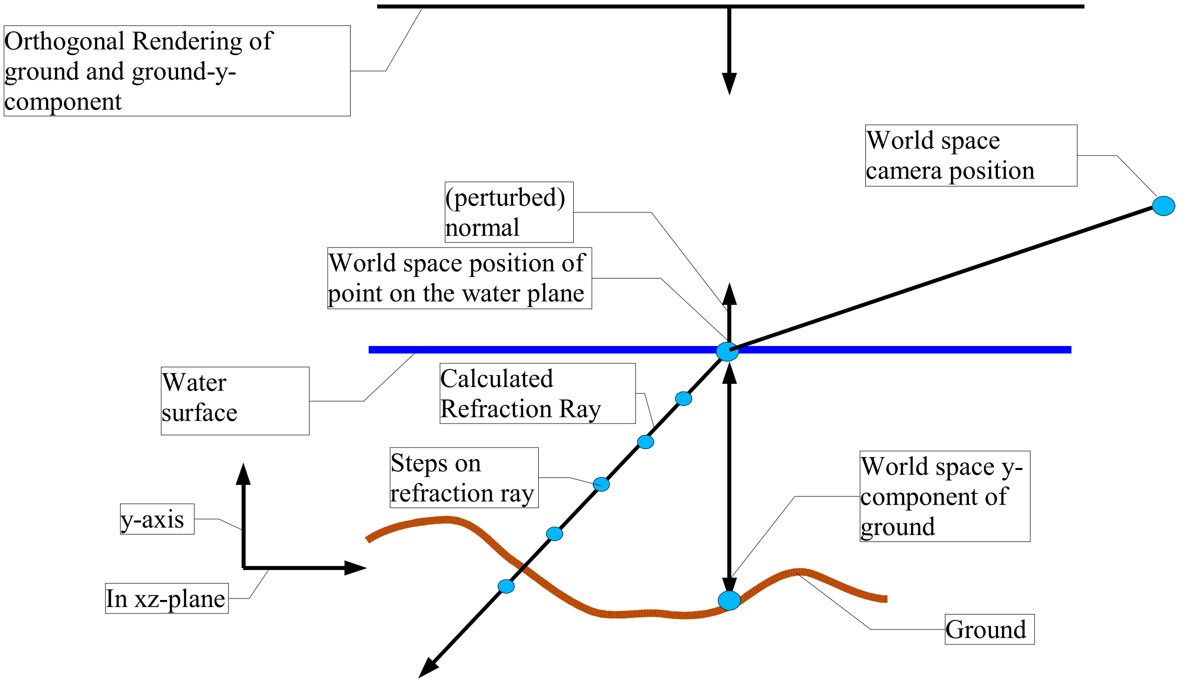
Markus Lipp, Student at Vienna University of Technology
markus-lipp@gmx.at

1. Introduction

This documentation focuses on the vertex and pixel shaders used for the water-surface rendering, as well on implementation details of the enclosing application. Section 2 will give a brief overview of the used technique, Section 3 covers the refraction water shader, Section 4 covers the shader with included reflections.

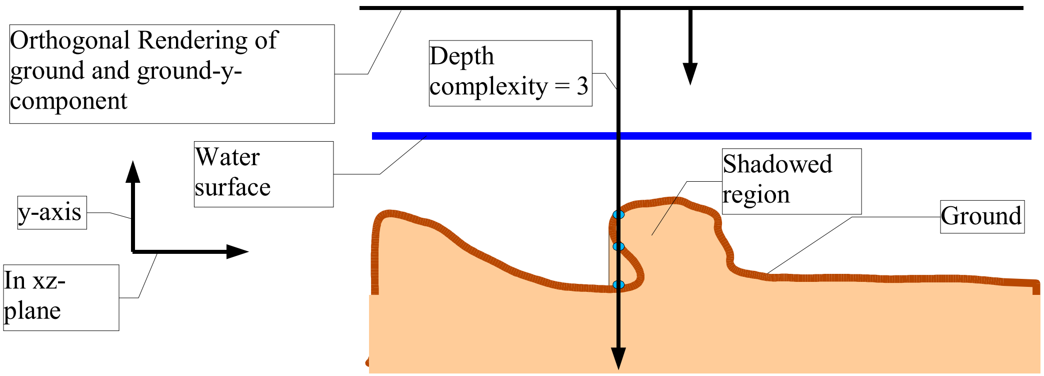
2. Method overview

The main idea of this method is to combine the water rendering method proposed in [1] with exact refraction calculation. Raycasting using steep parallax mapping as proposed in [2] is performed to get the refracted position on the ground. The following figure illustrates the geometry for this method:


Orthogonal Rendering of ground and ground-y-component
World space camera position
(perturbed) normal
World space position of point on the water plane
Water surface
Calculated Refraction Ray
Steps on refraction ray
y-axis
In xz-plane
World space y-component of ground
Ground

The following paragraph will briefly describe the used algorithm, please refer to Sections 3 for details.
At first, a texture containing the world-space y-positions of the ground, as well as a texture containing the colours of the ground are rendered from above the water surface. An orthogonal projection is used for this task. This has to be performed only once for static scenes. Of course these textures can also be predefined externally.
Then, during rendering of the water surface, the following steps are performed for each point on the water surface: Based on an vector to the camera position and the normal vector, the refraction vector is calculated. A loop is used to step along this refraction vector. For every step in the loop, the y-value of the ground below the current position is fetched from the world-space-y texture. Then the y-value of the current position is compared against the fetched y-position of the ground. If it is larger than the y-position of the ground, the ground has been crossed, and therefore the loop is terminated. Now an interpolation between the position before the crossing and the position after the crossing is performed, and the colour of the ground of this interpolated position is fetched. This is the refraction-colour we are looking for.

All calculations are performed in World space. During rendering of the water surface all information of the world geometry is obtained from the previously rendered orthogonal projection of the ground-y-component. To get colour information for a ground-position a previously rendered ground-texture is fetched. This results in the following limitation regarding the geometry of the ground: The depth complexity of the ground as seen from the orthogonal projection must not exceed 1, artefacts occur otherwise. This is caused by the fact that surfaces closer to the projection plane shadow surfaces further away. There is no way for the raycaster to get correct information for shadowed regions, therefore artefacts occur. The following figure illustrates this problem:



To understand where the water shaders come into play, let us have a look on how the water is rendered in the application. All function calls refer to the class `GL_Example`.

For every frame, `Draw()` is called. The left stereo camera is set up there and then `DrawSingleStereoImage()` is called. In this function the following steps are performed: `WaterPrepareTexturesMethod3()` prepares all textures necessary for water rendering. Then the scene is rendered with `DrawScene()`.

Afterwards `WaterDrawMethod3()` renders the water plane: At first the pixel shader is set. Then `WaterSetParameters()` transfers all necessary uniform water parameters to the shader. `WaterSetTextureProjectionMatrices()` sets up all Texture Projection Matrices required to project the rendered textures on the water plane. The following calls to `WaterBindTexturesMethod3()` and `WaterRenderQuad()` bind the textures and render the water quad.

All those steps are then performed for the right stereo image too.

2. Refraction Shader

As almost every pixel shader, the water pixel shader must be driven by an specific **vertex shader**. The vertex shader “water.vert” does this job. The most important parts of this shader are the varying variables to transfer a vector to the camera (`varyingVecEye`), a vector to the light (`varyingVecLight`) and the world position (`varyingVecWorldPosition`) of the actual point on the water plane to the pixel shader. Those variables are required in the pixel shader to calculate the refraction ray, the specular highlights and the starting position for the raycasting. The input variables `wave1-wave4` define the movement of the normalmap. Further input variables are `viewpos`, the world space position of the camera and `lightpos`, the world-space position of the light. Those input-variables have to be set by the application.
This is the sourcecode of the vertex-shader:

```
//translation vectors for normalmap texturecoordinates
uniform vec2 wave1;
uniform vec2 wave2;
uniform vec2 wave3;
uniform vec2 wave4;

//worldspace camera position
uniform vec3 viewpos;
//worldspace light position
uniform vec3 lightpos;

//vectors to be transferred to the pixel shader
varying vec3 varyingVecEye;
varying vec3 varyingVecLight;
varying vec3 varyingVecWorldPosition;

//Water plane transformations. Water plane must lie parallel to xz plane
//after transformations, else the depth comparisons are not correct.
uniform mat4 WaterTransformationMatrix;
```

```
void main(void)
{

        //Output Worldpos to pixel shader
        varyingVecWorldPosition = vec3(WaterTransformationMatrix*gl_Vertex);
        //get vector to worldspace view position, output to pixel shader
        varyingVecEye= viewpos - varyingVecWorldPosition.xyz;
        //get vector to worldspace light position, output to pixel shader
        varyingVecLight= lightpos - varyingVecWorldPosition.xyz;

        //Texture coordinates for Normalmap lookup
        gl_TexCoord[0] = vec4(gl_MultiTexCoord0.xy + wave1,0,1);
        gl_TexCoord[1] = vec4(gl_MultiTexCoord0.xy + wave2,0,1);
        gl_TexCoord[2] = vec4(gl_MultiTexCoord0.xy + wave3,0,1);
        gl_TexCoord[3] = vec4(gl_MultiTexCoord0.xy + wave4,0,1);

        //transform vertex coords
        vec4 vpos = gl_ModelViewProjectionMatrix * gl_Vertex;
        gl_Position = vpos;

}
```

There is one important thing to consider: The transformations of the water plane (which is assumed to lie on the xz-plane untransformed) must result in a water plane that is parallel to the xz-plane. If this is not the case, water depth-calculations in the pixel shader would not be correct, as the world-y-coordinate is used for this task. To make arbitrary transformations of the water surface possible, the correct distance to the water plane is required. Therefore depth-calculations would need to include a dot-product of the world-position with the plane-parameters of the water-surface .

The **pixel shader** requires various textures to work properly:

```
//Refraction Texture, Rendered from Top
uniform sampler2D refractionTexture;
//Reflection Texture, Cubemap containing skybox
uniform samplerCube reflectionTexture;
//Normal pertubation map, defined in water-plane
uniform sampler2D normalMap;
//Y-components of geometry under the water, rendered from top
uniform sampler2D worldYComponent;
//Fresnel Lookup Texture, lookup from costheta- > Transm. Coeff,
Refr. Coeff
uniform sampler2D fresnelLookup;
```

Let us have a detailed look on these textures:

`RefractionTexture`: This texture is rendered with an orthogonal projection with the view-vector perpendicular to the water surface, from above the water plane onto the waterplane. The texture should enclose the whole water plane, including a safety border around it, to prevent artifacts. **GL_Example::WaterDrawTextureRefractionTop()** is used to render this texture. This texture must only be updated if the scene changes. Of course this texture can also be a predefined external texture. **gl_TextureMatrix[0]** must contain the matrix for projecting this texture on to the water plane. **GL_Example::WaterSetTextureProjectionMatrices()** initializes this matrix.

`ReflectionTexture`: A simple cubemap including the skybox. **GL_Example::CreateCubeMap** creates this cubemap.

`NormalMap`: The normals of the water surface are stored in this texture, with each color encoding one normal vector axis (r=x, g=y, b=z). The range of the texture [0,1] is transformed to [-1,1] in the pixel shader. Normals should point upwards from the water plane, therefore g must be larger than 0.5. **w_normalmap.bmp** encodes such a normalmap.

`worldYComponent`: Just like RefractionTexture this texture is rendered from above the water-plane. It must encode the world-y-component of the rendered scene in the red-component. Floating point precision is required for this texture, nearest-filtering is preferred. **GL_Example::WaterDrawTextureDepthRefractionTop()** creates this texture.
This texture must only be updated if the scene changes, and it can be predefined as an external texture (for example a heightmap). **gl_TextureMatrix[0]** must contain the matrix for projecting this texture on to the water plane.
GL_Example::WaterSetTextureProjectionMatrices() initializes this matrix.

`fresnelLookup`: This textures stores the transmission coefficient in the red, and the refraction coefficient in the green channel. It is addressed with cos(N*V) as texture-coordinate, the acos-term is integrated into the texture. **GL_Example::CreateFresnelLookupTexture()** is used to create this texture. Floating point precision is the best choice for this texture, nearest-filtering is preferred.

The next part of the pixel shader defines the uniform variables:

```
//refractioncoefficient = c_air/c_water
uniform float refrCoeff;
//watercolor, for modulation of refracted color
uniform vec4 waterColor;
//color for modulation of reflected light
uniform vec4 reflectColor;
//color of specular highlight
uniform vec3 lightcolor;
//multiplicator to enhance normal-pertubation. 0=max pertubation
1=normal pertubation
uniform float BumptexMult;
```

Then the varying variables, as provided by the vertex shader are defined:

```
//vectors set up by the vertex shader
//World space position of point on the water-plane
varying vec3 varyingVecWorldPosition;
//vector from varyingVecWorldPosition to Worldspace Camera Position
varying vec3 varyingVecEye;
//vector from varyingVecWorldPosition to Worldspace Light Position
varying vec3 varyingVecLight;
```

Finally, all required resources are defined, let us move on to the pixel shader. At first the varying variables have to be renormalized:

```
void main (void)
{
        //Get vectors to eye&light
        //Renormalize interpolated varying vectors
        vec3 vEye=normalize(varyingVecEye);
        vec3 vLight = normalize(varyingVecLight);
```

When we have those vectors we start sampling the normalmap multiple times, then average it, as proposed in [1].

```
        //Sample normalmap at different locations
        vec3 vBumpTexA =  vec3(texture2D(normalMap, gl_TexCoord[0].xy));
        vec3 vBumpTexB =  vec3(texture2D(normalMap, gl_TexCoord[1].xy));
        vec3 vBumpTexC =  vec3(texture2D(normalMap, gl_TexCoord[2].xy));
        vec3 vBumpTexD =  vec3(texture2D(normalMap, gl_TexCoord[3].xy));

        //Average Bump Layers, transform from [0,1] to [-1,1]
        vec3 vBumpTex = normalize( 2.0 *
        (vBumpTexA+vBumpTexB+vBumpTexC+vBumpTexD) - 4.0);

        //Increase bumps: decrease y-component, then renormalize
        //This results in larger xz-components
        vec3 vBumpTexMore =  normalize( vBumpTex*vec3(1.0,BumptexMult,1.0));
```

Then the refraction-vector is calculated. Fortunately, OGLSL provides the function “Refract” to do this. This function takes a vector from the eye to the point (equal to -`vEye`), the normal and the refractrion coefficient as parameters.

```
        //Calc correct refraction  vector
        vec3 R = normalize(refract(-vEye, vBumpTexMore, refrCoeff));
```

As proposed in [2], the size of the raycasting steps is decreased for shallow angles of the ray direction. Shallow angles occur when the y-component of the Refraction-vector is close to zero. Therefore a small step-size is used for this case.

```
        //Calculate ideal stepsize: If R.y = 0 the reflection ray is at an
        gracing angle => use maximum steps=10
        //at R.y=1: use minimum steps=3
        float numsteps = mix(10,3,-R.y); //Linear Interpolate based on -R.y
```

```
float stepsize= 1.0 / numsteps;
//Scale Refraction vector with stepsize
R*=stepsize;
```

Now that `stepsize` is optimal, let us move on to the raycasting part.
`Depth` holds the world-space y-component of the current position along the refraction-ray, while `olddepth` holds the same for the ray-position of the previous step:

```
float depth, olddepth;
```

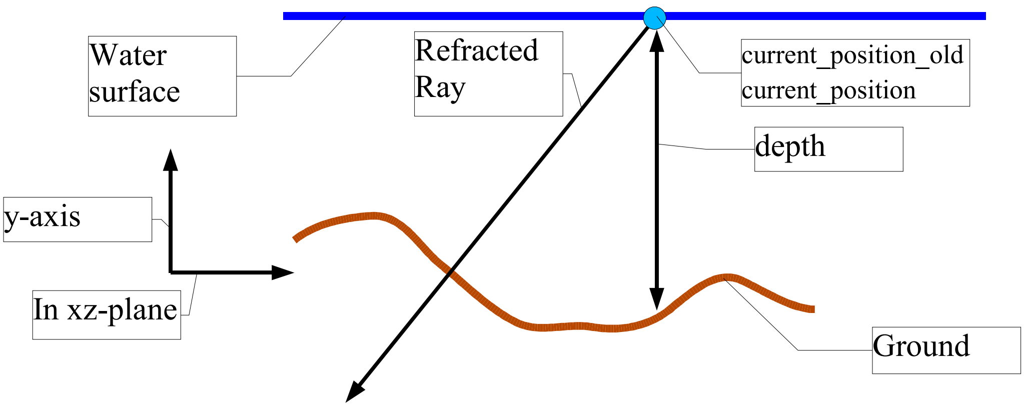
At first initializations for the first step are performed. `current_position` always stores the actual position on the refraction ray. `current_position_old` does this for the previous step.

```
//Start with approximation at vertex-world-position
vec3 current_position = vec3(varyingVecWorldPosition);
vec3 current_position_old=current_position;
olddepth=0;
```

Projective Texturing is used to obtain the depth (world-space y-position) of the scene point that lies below `current_position`. Texture coordinates are calculated with `gl_TextureMatrix[0]*vec4 (current_position,1)`. Note that `texture2D` is used instead of `texture2DProj`, as we have an orthogonal projection and therefore do not need a homogeneous divide.

```
//Fetch the depth of the water at the starting-position
vec4 depthtex = texture2D(waterDepth, ( gl_TextureMatrix[0]*vec4
(current_position,1)).xy);
depth = depthtex.r;
```

The current situation is illustrated in the following figure:



We then start the main loop. At first one step along the refraction ray is performed. The ground-depth is then fetched for this position. If the depth of the position on the ray is higher than the depth of the ground, we have crossed the ground and can therefore break the loop.

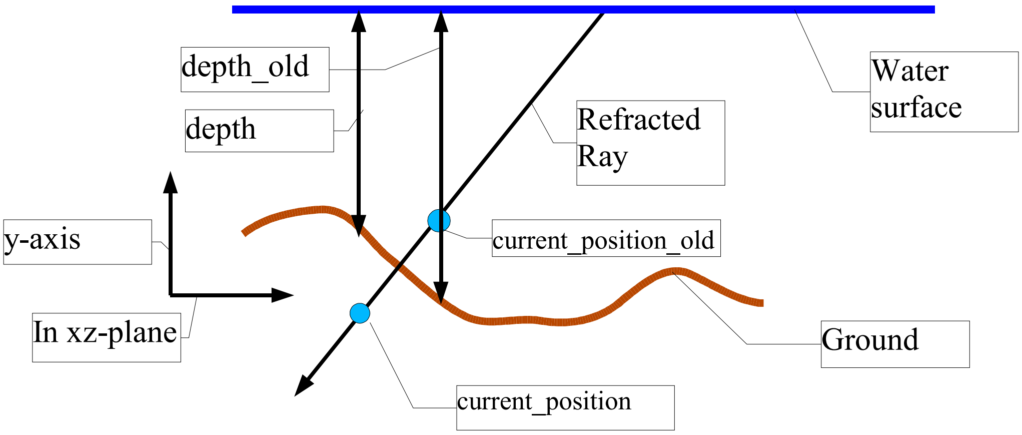
```
for (int i=0; i<40;i++)
{
      //Save old position on ray
      current_position_old=current_position;
      //step further on ray
      current_position +=R;
      //save old depth
      olddepth=depth;

      //Fetch ground-depth for current position on ray
      depthtex = texture2D(waterDepth,  ( gl_TextureMatrix[0]*vec4
      (current_position,1)).xy);
      depth = depthtex.r;

      //Assume water plane is parallel to xz-plane
      //-> a comparison of world-space y can be used for water depth
      comp.
      if (depth> current_position.y )
```

```
                        break;
        }
```

Afterwards we have to linearly interpolate between the position before we crossed the ground (`current_position_old`) and the position after this crossing (`current_position`) based on the previous depth of the ground (`olddepth`) and the current depth of the ground (`depth`). The following figure illustrates this problem:



The result is characterized trough a linear interpolation defined by the equation

$$f(t)=currentposition_{old}\cdot t+currentposition\cdot(1-t), t\in[0,1]$$

`t` is derived the following way: Let us abbreviate `current_position_old.y` with `cpoy` and `current_position.y` with `cpy`. We want the value `t` for the intersection of the line between `cpoy` and `cpy` and the line between `depth_old` and `depth`. Those two lines are defined trough the following equations:

$$y_1(t)=depth_{old}+(depth-depth_{old})\cdot t$$
$$y_2(t)=cpoy+(cpoy-cpy)\cdot t$$

Combining those equation leads to:

$$depth_{old}+(depth-depth_{old})\cdot t=cpoy+(cpoy-cpy)\cdot t$$

Then we resolve for `t`:

$$t=(cpoy-olddepth)/(cpoy-cpy+depth-olddepth)$$

This leads to the following pixel shader code:

```
float cpoy=current_position_old.y;
float cpy = current_position.y;

current_position=mix(current_position_old,current_position,(cpoy-
   olddepth) / (cpoy-cpy+depth-olddepth));

//Fetch Interpolated Refracted Position
vec4 vRefraction = texture2D(refractionTexture,  ( gl_TextureMatrix[0]
   *vec4(current_position,1)).xy);
```

The distance the ray travelled trough water should affect the refraction color `vRefraction`. Therefore we interpolate between the watercolor and the refracted color based on this distance, using an (arbitrary) linear factor of 1/7.5:

```
//Calculate distance Ray traveled trough water
float travelLength = length(current_position-varyingVecWorldPosition);
//travelLength==7.5 => 1, travelLength=0 => 0
travelLength = clamp( travelLength/7.5, 0,1);
//Calc new attentuated vRefraction
vRefraction=mix(vRefraction, waterColor, travelLength);
```

Finally, everything refraction specific is calculated. Let us move on to the reflection part. If `NOREFLECTION` is defined, the shader is compiled with reflections disabled.

```
    #ifndef NOREFLECTION

        //Calculate reflection Vector
        R = normalize(reflect(-vEye, vBumpTexMore));
        //clamp y to 0.0001, else reflection vector would go into water
        R.y = max(R.y,0.001);
        //Fetch from cubemap, modulate with reflectColor
        //Somehow y-component is inverted, so it is rescaled with -1.
        probably a bug in my code.
        vec4 vReflection = textureCube(reflectionTexture,  normalize(R)*vec3
        (1,-1,1) )*reflectColor;
```

Note that this results in no local reflection, as the cubemap only stores the skybox. Exact local reflection needs raycasting too, this is covered in the next section.
Specular Highlights based on the phong specular reflection model with arbitrary parameters are then calculated:

```
        //Calculate Specular reflection

        //Calculate specular Reflection vector
        vec3 rLight = dot(2*vBumpTex,vLight)*vBumpTex-vLight;
        //evaluate phong specular reflection model, modulate with lightcolor
        vec4 specular = vec4(pow(clamp(dot(rLight, vEye),0,1),256)
        *lightcolor,1);
        //multiply with specular strength (2)
        vReflection+=specular*2;
```

Finally, lookup for fresnel coefficients is performed based on cos(phi)=NdotV where phi is the angle between N and V. Then the reflection and the refraction colours are combined based on these coefficients.

```
        //Fresnel
        float NdotV =  clamp( dot(vEye, vBumpTex),0,1);

        //Lookup exact Fresnel Coefficients.
        vec4 FresnelCoeffs = texture2D(fresnelLookup,  NdotV) ;

        gl_FragColor =
        vReflection*FresnelCoeffs.g+vRefraction*FresnelCoeffs.r;

    #else
        //No Reflection
        gl_FragColor=vRefraction;
    #endif
```

3. Reflection and Refraction Shader

Unlike the previous shader, this shader also raycasts reflections for correct local reflections. This shader is very similar to the previous one, therefore only dissimilarities are described.
Two additional Textures are required, they are analogous to the refraction textures:

```
    //Y-components of geometry above the water, rendered from bottom
    uniform sampler2D worldYComponentReflection;
    //Reflection rendered orthogonal and perpendicular to water surface
    uniform sampler2D reflectionBottom;
```

`worldYComponentReflection:` This texture is rendered with an orthogonal projection with the view-vector perpendicular to the water surface, from below the water plane onto the waterplane. It must encode the world-y-component of the rendered scene in the red-component. Floating point precision is required for this texture, nearest-filtering is preferred. This texture should enclose the whole water plane.
GL_Example::WaterDrawTextureDepthReflectionBottom() is used to render this texture. This texture must only be updated if the scene changes. **gl_TextureMatrix[1]** must contain the matrix for projecting this texture on to the water plane. **GL_Example::WaterSetTextureProjectionMatrices()** initializes this matrix.

`reflectionBottom`: This texture is rendered with an orthogonal projection with the view-vector perpendicular to the water surface, from below the water plane onto the waterplane. The texture should enclose the whole water plane.
GL_Example::WaterDrawTextureReflectionBottom() is used to render this texture. This texture must only be

updated if the scene changes. **gl_TextureMatrix[0]** must contain the matrix for projecting this texture on to the water plane. **GL_Example::WaterSetTextureProjectionMatrices()** initializes this matrix.

The reflection vector is calculated the same way.

```
	#ifndef NOREFLECTION

		//Calculate reflection Vector
		R = normalize(reflect(-vEye, vBumpTexMore));
		//clamp y to 0.0001, else reflection vector would go into water
		R.y = max(R.y,0.001);
		R=normalize(R);
```

Then raycasting is initialized just like for the refraction. At first an ideal `stepsize` is calculated:

```
		//Calculate ideal stepsize: If 1-R.y = 1 the reflection ray is at an gracing angle
		numsteps = mix(0.5,4,1-R.y);
		stepsize= 1.0 / numsteps;
		//scale Reflection ray with stepsize
		R*=stepsize;
```

The starting condition is set:

```
		//Trace Ray!

		//Start at vertex position
		current_position = varyingVecWorldPosition;
		current_position_old=current_position;
		olddepth=0;
		depthtex = texture2D(worldYComponentReflection, (gl_TextureMatrix[1]
		*vec4(current_position,1)).xy);
```

The world-y-component is in the red channel.

```
		depth = depthtex.r;
```

Here is one main difference: `noHit` stores whether we hit something during tracing. If we do not hit anything, we just use the cubemap for reflection-colour.

```
		bool noHit=true;
```

Main tracing loop:

```
		for (int i=0; i<60;i++)
		{
			current_position_old=current_position;
			//Step further
			current_position +=R;
			//calculate texcoords for current position
			//Assume Water.y = 0
			vec2 texcoord= (gl_TextureMatrix[1]*vec4(current_position,1)).
			xy;
```

If we are outside the water-surface, we stop tracing, and assume that there is no hit:

```
			//Outside of reflection texture? -> no hit!
			if ((texcoord.x<0)||(texcoord.x>1)||
				(texcoord.y<0)||(texcoord.y>1))
			{
				break;
			}

			//Fetch depth
			depthtex = texture2D(worldYComponentReflection, texcoord);
			olddepth=depth;
			depth = depthtex.r;
```

```
        //If we are below the reflection surface, we have a hit
        if (depth> current_position.y)
        {
            noHit=false;
            break;
        }
```

Also quit when we are to high above water. The value 0.8 must be set according to the maximum height of the reflected scene.

```
        //end if to high over water surface
        if (current_position.y>0.8)
        {
            break;
        }
    }

    vec4 vReflection;
```

No hit results in a color-fetch from the cubemap. Otherwise the same interpolation as derived for refractions is performed.

```
    if (!noHit)
    {
        //Interpolate, one step of zero-approximation
        float cpoy= current_position_old.y;
        float cpy = current_position.y;
        current_position=mix(current_position_old,current_position,
        (cpoy-olddepth) / (cpoy-cpy+depth-olddepth) );
        vReflection=texture2D(reflectionTop,  (gl_TextureMatrix[1]
        *vec4(current_position,1)).xy)*reflectColor;
    }
    else
    {
        //No hit with reflection surface -> just fetch color from
        cubemap
        //otherwise fetch color from reflection-texture
        vReflection = textureCube(reflectionTexture,  normalize(R)
        *vec3(1,-1,1) )*reflectColor;
    }
```

The rest of this shader is equal to the previous one.

References

[1] Tiago Sousa (Crytek). *Generic Refraction Simulation,* in GPU Gems 2, ISBN: 0321335597, Published: Mar, 2005, Addison Wesley

[2] Morgan McGuire and Max McGuire. *Steep parallax mapping.* I3D 2005 Poster.