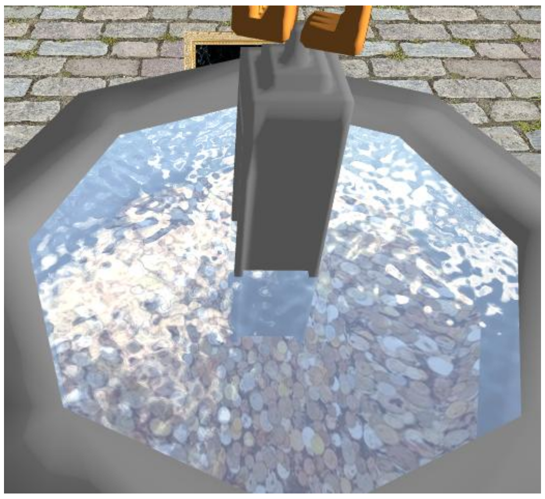
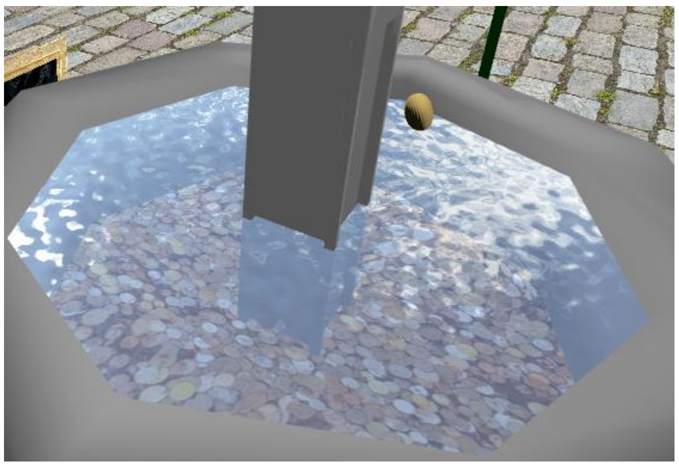
Special Effect: Animated water surface

Introduction

Since implementing a good looking animated water surface with WebGL is very interesting in our opinion, it was clear to take this special effect into account for our movie. To satisfy all specifications for the water surface, it was essential to compute correct water reflections of the world's skybox as well as some realistic wave movements. The keyword for implementing the water's reflection is "texture rendering". Respectively, the wave movement algorithm relies on a concept called "DuDv Texture Mapping".

Of course the reflection should be computed correctly with respect to the camera position. Therefore it is necessary to refresh the reflection texture in every render step.

Dependencies

There occurred a problem concerning reloading reflection texture on resizing the browser window. If the size of the browser window is changed, the texture which provides the base of the reflection image has to be resized as well. This resizing operation is done with a Javascript event.

```
//On window resize, we have to set new attributes for waters reflection textures
window.addEventListener ('load', function () {
    window.onresize = function (){
        setRenderTextureAttributes();
    }
});
```

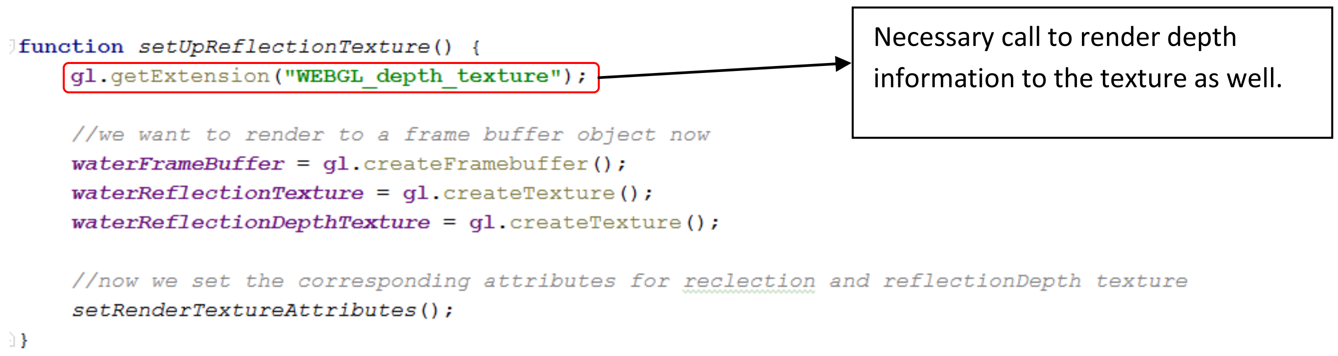
This function can be executed in Google Chrome without any problems. But Mozilla Firefox doesn't trigger this function sometimes. That's the reason why the reflection texture is probably not displayed correctly at resize events in Firefox.

Algorithm Description

Reflection Texture

In order to compute a good looking reflection on the well’s water surface, we tried to handle this with rendering to a texture. To initialize this, some basic setup has to be done:

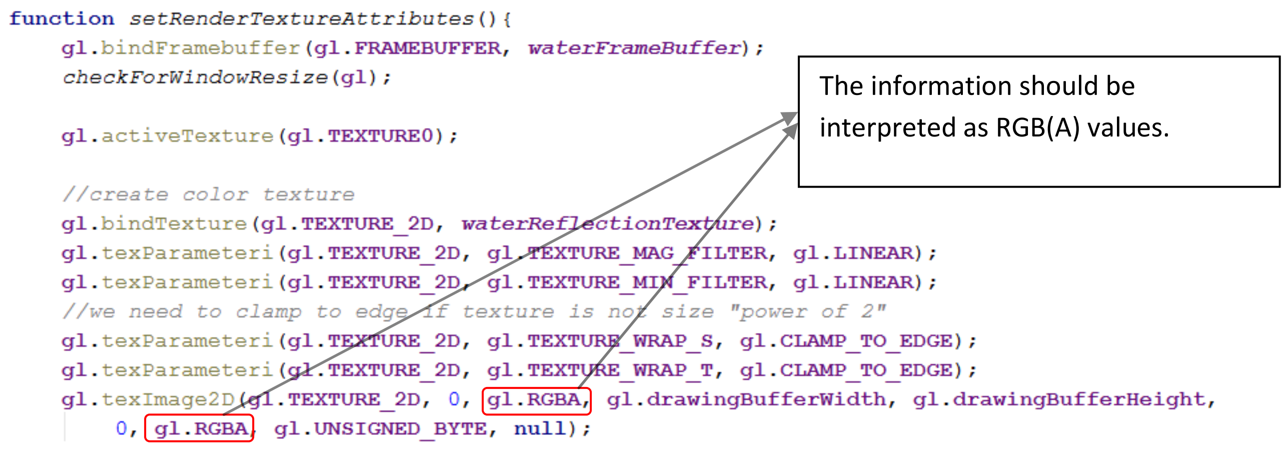
At first we want to record some parts of the current displayed frame and render these parts onto the reflection texture. Since we don’t want to render the recorded fragments on the screen, we have to create our own framebuffer object where we can render to. After recording we can put the content of the framebuffer into a texture object. After these operations we put this rendered texture onto the water’s quad object.



```
function setUpReflectionTexture() {
    gl.getExtension("WEBGL_depth_texture");

    //we want to render to a frame buffer object now
    waterFrameBuffer = gl.createFramebuffer();
    waterReflectionTexture = gl.createTexture();
    waterReflectionDepthTexture = gl.createTexture();

    //now we set the corresponding attributes for reclection and reflectionDepth texture
    setRenderTextureAttributes();
}
```

Necessary call to render depth information to the texture as well.

This function creates the necessary framebuffer object, the reflection texture and the corresponding depth texture to provide depth test. Furthermore, we call the next function `setRenderTextureAttributes()`, which does the basic configuration for the created instances.



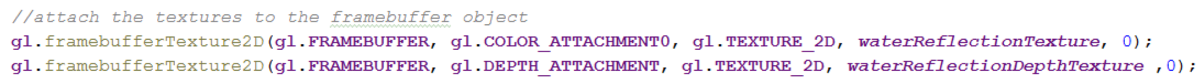
```
function setRenderTextureAttributes(){
    gl.bindFramebuffer(gl.FRAMEBUFFER, waterFrameBuffer);
    checkForWindowResize(gl);

    gl.activeTexture(gl.TEXTURE0);

    //create color texture
    gl.bindTexture(gl.TEXTURE_2D, waterReflectionTexture);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
    //we need to clamp to edge if texture is not size "power of 2"
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.drawingBufferWidth, gl.drawingBufferHeight,
        0, gl.RGBA, gl.UNSIGNED_BYTE, null);
```

The information should be interpreted as RGB(A) values.

Firstly, we declare our created framebuffer object as the currently activated one. Afterwards, the initial texture stuff has to be done. The `checkForWindowResize()` function is called here because we have to take the resizing aspect of the browser window under consideration too. This is followed by the standard texture property stuff. Important here is, that we have to clamp the rendered images since we usually do not work with “power of 2” image data. Almost the same commands have to be done for the depth texture, which we need as well.

Except the `gl.RGBA` parameters have to be substituted with the `gl.DEPTH_COMPONENT` keyword.

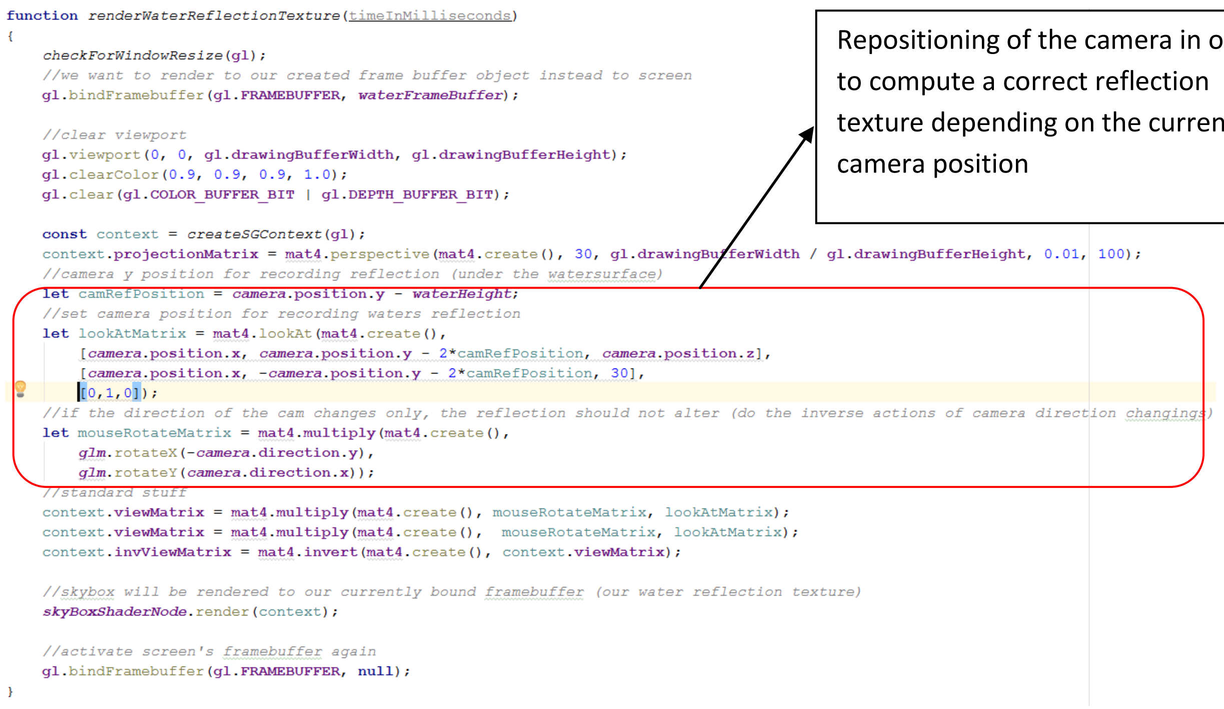
Now we have to combine our framebuffer object with the textures we want to render to. This can be implemented as follows:



```
//attach the textures to the framebuffer object
gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0, gl.TEXTURE_2D, waterReflectionTexture, 0);
gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.DEPTH_ATTACHMENT, gl.TEXTURE_2D, waterReflectionDepthTexture ,0);
```

After these operations, finally, we have to unbind our framebuffer to activate the screen’s framebuffer again.

Now the basic setup for the rendering part is over. Let’s continue with the interesting “render to texture” part.

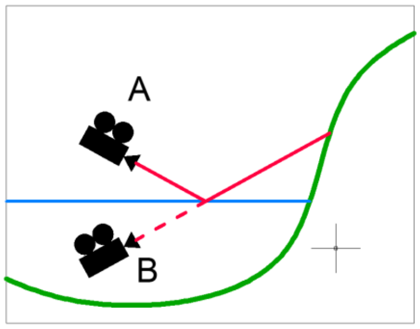


```
function renderWaterReflectionTexture(timeInMilliseconds)
{
    checkForWindowResize(gl);
    //we want to render to our created frame buffer object instead to screen
    gl.bindFramebuffer(gl.FRAMEBUFFER, waterFrameBuffer);

    //clear viewport
    gl.viewport(0, 0, gl.drawingBufferWidth, gl.drawingBufferHeight);
    gl.clearColor(0.9, 0.9, 0.9, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    const context = createSGContext(gl);
    context.projectionMatrix = mat4.perspective(mat4.create(), 30, gl.drawingBufferWidth / gl.drawingBufferHeight, 0.01, 100);
    //camera y position for recording reflection (under the watersurface)
    let camRefPosition = camera.position.y - waterHeight;
    //set camera position for recording waters reflection
    let lookAtMatrix = mat4.lookAt(mat4.create(),
        [camera.position.x, camera.position.y - 2*camRefPosition, camera.position.z],
        [camera.position.x, -camera.position.y - 2*camRefPosition, 30],
        [0,1,0]);
    //if the direction of the cam changes only, the reflection should not alter (do the inverse actions of camera direction changings)
    let mouseRotateMatrix = mat4.multiply(mat4.create(),
        glm.rotateX(-camera.direction.y),
        glm.rotateY(camera.direction.x));
    //standard stuff
    context.viewMatrix = mat4.multiply(mat4.create(), mouseRotateMatrix, lookAtMatrix);
    context.viewMatrix = mat4.multiply(mat4.create(),  mouseRotateMatrix, lookAtMatrix);
    context.invViewMatrix = mat4.invert(mat4.create(), context.viewMatrix);

    //skybox will be rendered to our currently bound framebuffer (our water reflection texture)
    skyBoxShaderNode.render(context);

    //activate screen's framebuffer again
    gl.bindFramebuffer(gl.FRAMEBUFFER, null);
}
```

Repositioning of the camera in order to compute a correct reflection texture depending on the current camera position

At beginning we again check for a window resize and set our framebuffer object to the currently used one in order to render to this framebuffer instead of the screen’s framebuffer. These operations are followed by some standard operations like clearing viewport and set projection matrix.

But now it becomes interesting. Since we want to render an almost correct reflection onto our water surface, we have to relocate our camera.



Assume that our camera is currently at position A. To get the correct reflected light rays. We have to change the location of our camera for the reflection render step like it is illustrated in the picture.

Source: http://trederia.blogspot.co.at/2014/09/water-in-opengl-and-gles-20-part3.html

Calculate camera height with respect to water height:

$$relativeCameraHeight = cameraPosition\ y - waterHeight$$

Relocate the camera “under the water”:

$$reflectionCameraHeight = cameraPosition\ y - 2 * relativeCameraHeight$$

Finally, it is needed to change the camera’s “looking at” vector:

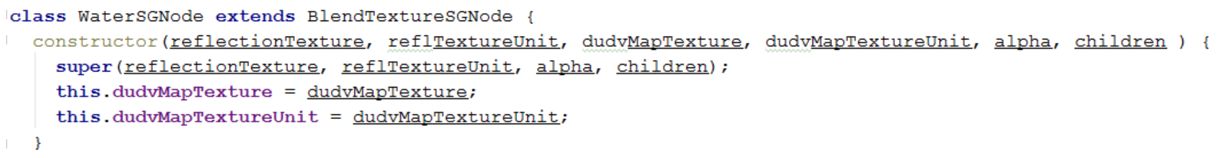
$$reflectionCamera\ y\ Direction = -camera\ y\ Direction$$

With these mathematical calculations we repositioned the camera to record the correct reflection texture for the corresponding camera position.

After that, the different transformation matrices are set as usual. After all this operations and commands the `render()` function can be called.

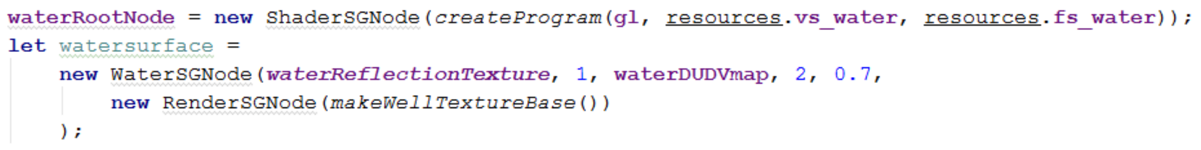
Finally, set the screen’s framebuffer as the current one.

With the class `WaterSGNode` (which inherits from `BlendTextureSGNode`) we upload some important properties to the water vertex shader and water fragment shader.



```
class WaterSGNode extends BlendTextureSGNode {
  constructor(reflectionTexture, reflTextureUnit, dudvMapTexture, dudvMapTextureUnit, alpha, children ) {
    super(reflectionTexture, reflTextureUnit, alpha, children);
    this.dudvMapTexture = dudvMapTexture;
    this.dudvMapTextureUnit = dudvMapTextureUnit;
  }
```

By extending `BlendTextureSGNode` we upload an additional alpha uniform since we want to enable blending in order to make the water transparent.



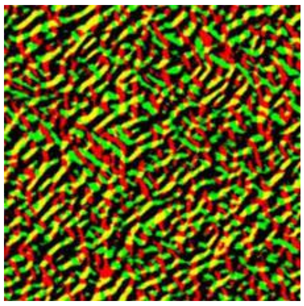
```
waterRootNode = new ShaderSGNode(createProgram(gl, resources.vs_water, resources.fs_water));
let watersurface =
    new WaterSGNode(waterReflectionTexture, 1, waterDUDVmap, 2, 0.7,
        new RenderSGNode(makeWellTextureBase())
    );
```

This code creates the well’s water surface and transfers the reflection texture, the dudv texture and the waters alpha value to the water shaders.

DuDv Texture Mapping

Computing the reflection texture is just the half of work. We want to introduce some nice wave effects onto the reflection texture as well. This is done with an approach which is called “DuDv Mapping”.

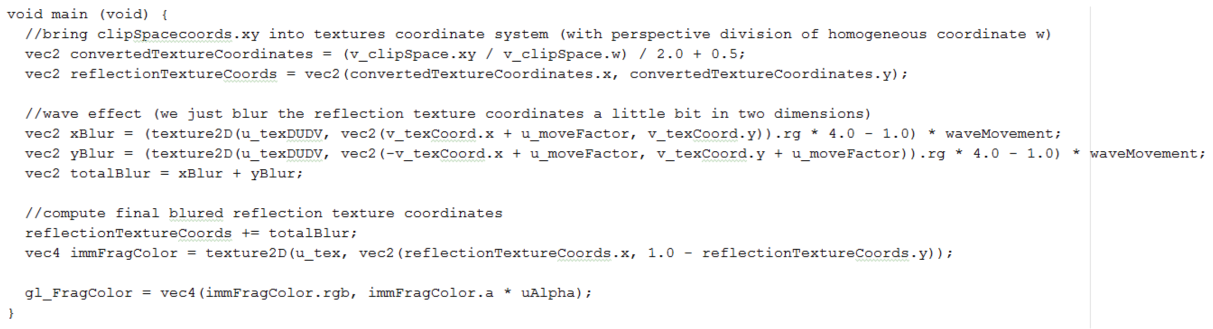
A DuDv map is a simple image file which stores direction vectors in x- and y direction. With the help of this information, the reflection texture can be distorted slightly. Since only the x and y coordinates of the texture are used the map only has green/red/yellow color components…



The whole distortion part is done in the waters fragment shader.

Shader Code

Because just standard stuff is made in `water.vs.glsl`, we skip to the waters fragment shader `water.fs.glsl`.



```
void main (void) {
  //bring clipSpacecoords.xy into textures coordinate system (with perspective division of homogeneous coordinate w)
  vec2 convertedTextureCoordinates = (v_clipSpace.xy / v_clipSpace.w) / 2.0 + 0.5;
  vec2 reflectionTextureCoords = vec2(convertedTextureCoordinates.x, convertedTextureCoordinates.y);

  //wave effect (we just blur the reflection texture coordinates a little bit in two dimensions)
  vec2 xBlur = (texture2D(u_texDUDV, vec2(v_texCoord.x + u_moveFactor, v_texCoord.y)).rg * 4.0 - 1.0) * waveMovement;
  vec2 yBlur = (texture2D(u_texDUDV, vec2(-v_texCoord.x + u_moveFactor, v_texCoord.y + u_moveFactor)).rg * 4.0 - 1.0) * waveMovement;
  vec2 totalBlur = xBlur + yBlur;

  //compute final blured reflection texture coordinates
  reflectionTextureCoords += totalBlur;
  vec4 immFragColor = texture2D(u_tex, vec2(reflectionTextureCoords.x, 1.0 - reflectionTextureCoords.y));

  gl_FragColor = vec4(immFragColor.rgb, immFragColor.a * uAlpha);
}
```

At first some conversion has to be done because the textures coordinate system differs from the vertex coordinate system. Additionally, the perspective division is made to display the mapped texture correctly with respect to perspective. This is followed by the actual part which computes the distortion of the water’s surface. By the way, it’s worth mentioning that we use an uniform variable called `u_moveFactor` which changes by time and is computed in the Javascript code. With the help of this factor we can do this “moving wave” effect.

`xBlur` and `yBlur` are the two distortion factors. Both of them are simple DuDv texture lookups with some basic mathematical calculations which are influenced by `u_moveFactor`.

To have a realistic distortion over the water surface, distortion in reflection textures x- and y direction have been implemented. This two blur vectors can be added to the default, already converted texture coordinates of the reflection texture.

Afterwards the reflection texture lookup is done (the y coordinate has to be flipped to get expected texels from texture coordinate system).

Finally, multiply the alpha channels with the uniform alpha value to make our water transparent.

That's it!