REAL-TIME WATER SIMULATION

Jagjeet Singh Dhaliwal (2008CS50212)
cs5080212@cse.iitd.ac.in

CSD750, Minor Project
Supervised by: Dr. Prem Kalra

ABSTRACT

Real-time simulation of liquids like water is an important task nowadays in the field of computer graphics. In this project report, I will give a detailed description of how I achieved the challenging task of simulating a water body. My goal is to be able to render water while giving more importance to the visual appearance rather than the physical simulation. So we will explore methods different than the standard Navier-Stokes Equation which is normally used to describe the motion of incompressible fluids like water. I will first give a description of the methods I have implemented and then go about describing how I implemented them. Results of the simulation are shown at the end.

INTRODUCTION

For the minor project, I have been working on the problem of Real-time Water Simulation. The process of rendering water in real-time is highly dependent on how much realism is needed in an application which can be a computer game, a movie or just some scientific setup being used to run some simulations. In the early days of computer graphics, water basically used to be treated as a planar surface. But with the drastic improvement in the rest of the computer graphics, the need for better animation of water surfaces grew strong.

So what's different about water that it is considered to be so interesting in computer graphics? Rendering water is quite different from objects we usually render. Water is a dynamic fluid so we have to update it every frame in our simulations. Many initial approaches used the very scientifically accurate Navier-Stokes equation to represent the water surface. But this involves a lot of heavy-weight computation. Since then many approximations have been introduced so as to capture the water surface as accurately as possible in real-time but also to make sure that the implementation is computationally feasible. Our main requirements are that the simulation should be cheap to compute and the memory computation should be low so that we could run it even on consoles with low memory usage with the rendering being visually plausible. Some techniques have also been introduced to make the surface interactive but we will steer clear from this complexity by concentrating mostly on the visual aspect of water. We have to keep in mind that the look of the water depends mostly on how it interacts with the environment. So for this project, I have implemented two scenes. One of which is fairly simple and low on visual realism. This scene is that of a lake. The other scene is however more realistic and the one which I spent the most time on. This more realistic scene is that of a larger water body. Now first we will know more about the theory behind water animation which one can skip if one is already well versed with techniques used in water simulation. After that I will describe the methods I have implemented.

SECTION 1: CONCEPTS (Skip if already familiar)

In this section we will go through the concepts involved based on the past work that used them. You can skip this if you are already familiar with them.

1.1 WHAT IT INVOLVES

The complex task of water simulation can be easily understood if we can break it down into subparts.

1. Surface representation – What we use most commonly to represent water are Grids and Particle systems. In this project I have concentrated on using grids.

2. Height Field Generation – This involves various ways to describe the complex or simple water waves.

3. Reflection and Refraction rendering technique.

4. Fresnel term and rendering other various phenomena to make the scene more realistic.

1.1.1 **Surface Representation**

There have been techniques that have been used in the past to generate the surface grid so as to suit the requirements of the applications. The schemes used to make the surface grid makes us help chose the points on the grid on which to apply the height-field function. This is necessary because there are infinite points on the water surface and it won't be feasible to apply the height-field function to every single one of them. Some such techniques to represent the surface are mentioned in brief detail below:-

1. *Using 3D Grids* – This is perhaps seemingly the most obvious choice to represent a 3D fluid like water. All the physical forces are evaluated for each point in the grid. This can be very precise but also involves quite heavy-weight computation as well. It is normally used for small water bodies along with using NSE equation to generate the height map (as in reference 1).

2. *Using 2D Grids* – The use of 2D grids isn't as accurate as the use of 3D grids but it does provide us with a much simpler way to represent the water surface. The concept behind using a 2D grid is to represent the surface basically using a height field function.

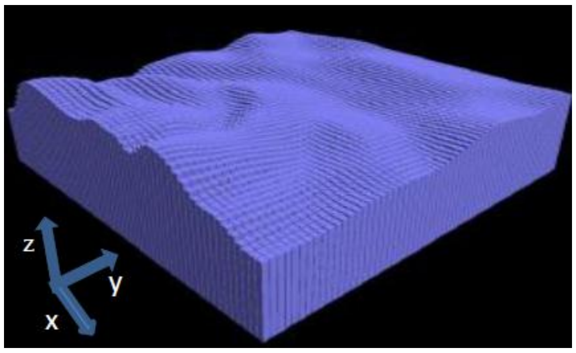


Figure 1: Representing a fluid surface as a 2D function H(x,y)

As I mentioned earlier that initially in games etc. water used to be just rendered as a plane surface which resulted in very low realism. To depict the complex movements at the surface of the water, the concept of a height-field fluid was introduced. In this concept the height of a fluid is represented as a function of two parameters x and y.

$$h_{surface}(x,y) = h_{plane}(x,y) + H(x,y).N_{plane}$$

The function H(x,y) is called the height-field function which decides how the surface will vary with x and y. is a unit vector in the direction perpendicular to the xy-plane. One of the major drawbacks of using this approach is that there can't be two height-field values for the same x and y which will make it impossible for us to render complex waves where the z values of any two points on the surface are the same.

The simplest way will be to use a simple symmetric rectangular grid to represent the surface but that way we will be giving the same detail to parts of the water that is far away too wasting some significant computation time while creating an effect that won't be as realistic. A popularly technique used to rectify this is the LOD (level-of-detail) scheme.

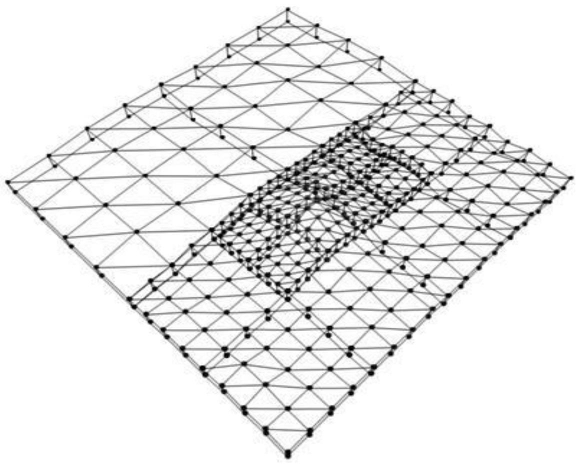


Figure 2: An example spatial representation of the surface grid according to LOD scheme.

The main principle behind this concept is that a large object could be divided into parts with several levels of detail. The farther the position of the patch from the viewer the lower is the level of detail associated with it. But the main problem occurs with this approach is when the level-of-detail switches from one patch to another causing some artifacts. (see reference 2).

Another popular scheme used to represent this grid is the Projected Grid Method (PGM) (see reference 3). Projected Grid approach is different from the other grid based methods as it tries to place the grid smoothly in the camera space itself. In this algorithm, first of all we create a regular rectangular grid in the post-perspective space. This grid is ensured to be orthogonal to the viewer. Then we project this grid onto the desired plane before transforming it back to world-space. Now that we have the grid in world-space, we apply the height-field function to the points on the grid and then render the resultant projected plane.

A real life analogy given in the thesis (see reference 3) is if you put a paper with a dotted grid on it in front of the spotlight and project it onto the surface.

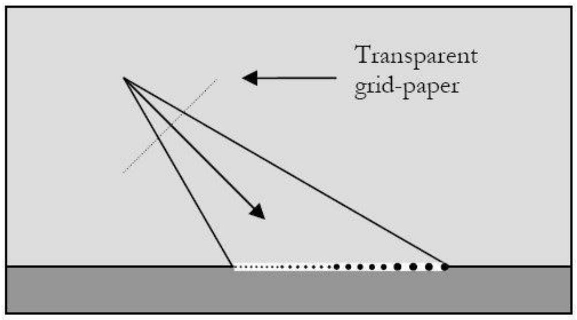


Figure 3: A Real-life Analogy of PGM.

This method backfires when the camera is not aiming towards the plane. Many points not supposed to be in the view also become incorporate in the viewing volume.

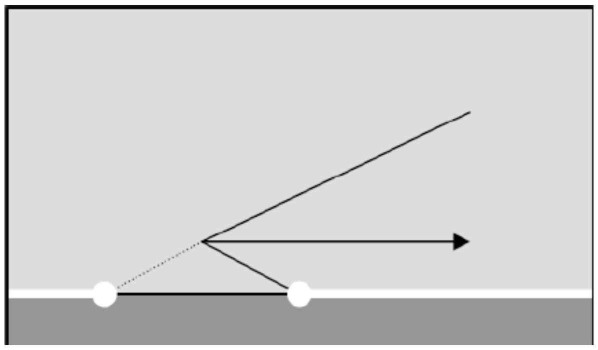


Figure 4: The 'backfiring problem'

So Claes introduced the concept of a projector. The projector is placed at the same location as the camera and is pointed in the same direction as long as the camera is aiming towards the plane. But now we have the flexibility to change its position and viewing direction whenever the camera backfires. In short, the projector is always aimed towards the plane no matter where the camera is aimed at thus solving our problem.

One important question that you may ask is why should not we simply treat the lines as rays and get rid of the extra points directly. The answer to this is that we are trying to avoid as many per vertex operations as possible and getting rid of them one by one in every frame wil lead to way too many of per-vertex operations.

1.1.2 Height Field Generation

The following are the methods that I feel need to be elaborated upon as it shows why I chose the methods I did to generate Height maps for the surface grid.

1. **Navier-Stokes Equation:** This is perhaps the most realistic equation out there to represent the physical simulation of incompressible viscous fluids like water. Forces like gravity, pressure and viscosity are incorporated in the equation. Needless to say this involves a lot of computation and is not computationally feasible for our implementation as this requires the entire grid to upgrade at the same time not just the visible volume.

2. **2D Wave equation:** This is an approximation used quite frequently to represent the surface of the water. This equation basically implies that acceleration of a point up and down is proportional to how quickly the steepness of the surface is changing (See reference 4).

3. **Coherent noise using Perlin Noise:** Random noise can also used to generate a height map for the height field fluid (See reference 5). Introduced by Ken Perlin, this method provides us with a continuous noise which is very much alike or similar to random noises found in nature. A simple 2D perlin noise won't help us much so what we use instead is multiple octaves of Perlin noise which will sum up to a fractal noise which is in fact very close to the random phenomenon in nature.

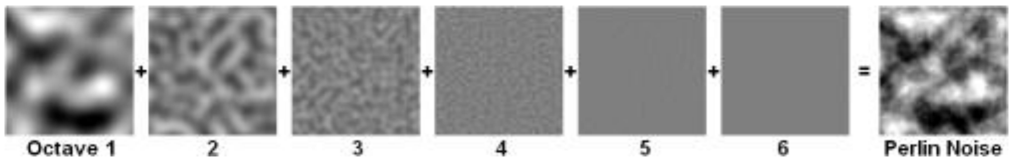


Figure 5: Sum of octaves to generate a fractal noise

The frequency of layer 1 is double of that of layer 2 and layer 3 is double that of layer 2 and so on. This is the reason that these layers are referred to as octaves (See reference 6). It is easily the most computationally efficient of the methods that we have discussed here.

4. **Fast Fourier Transform (FFT):** In this model, the height of the wave is assumed to be a function of time and the coordinates of the base plane. The height can then be found through the sum of various sine waves which can have different amplitudes and phases. This sum can be quickly calculated through the use of FFT. This method is tileable just like the perlin noise method and thus we can store it in a texture. In this method (See reference 7), first I initialize a 2D array with uniformly distributed random numbers. Then taking the 2D FFT of this array will give us a complex noise. After this we apply a 1/f^p (f = frequency and p is the smoothness factor of the resulting height map) filter on the transformed components. We can later revert back to the image space by applying Inverse Fast Fourier Transform. For a detailed description see http://www.keithlantz.net/2011/11/ocean-simulation-part-two-using-the-fast-fourier-transform/

1.1.3 Reflection and Refraction rendering techniques

It is important to understand how light behaves when it interacts with water in reality because that is what mainly decides how it is going to look to someone viewing it.

Rendering Reflection

Rendering reflections can be divided into two parts – Rendering the static environment and rendering the moving objects.

Rendering the static environment mainly incorporates the use of a pre-calculated environmental cube map to calculate the reflected colors. Cube maps are generally used to fake the appearance of environmental reflections. (See reference 8).

Rendering the moving objects is however an entirely different issue. We can't use pre-calculated environmental cube maps because they are always valid just for that particular point from which they were rendered. Rendering a cube map for moving objects every frame will be very costly and therefore is avoided. The solution to this will be to use screen-space reflections (See reference 4). In this method we assume the surface to be acting as a mirror. We reflect the world onto the other side of the water surface and store it into a 2D reflection texture. We then apply this texture is applied to the water surface by projective texturing.

Rendering Refraction

Rendering refractions is very much alike the process of rendering reflections (See reference 4). A refraction map is generated just like above but this time we render everything that is below the surface unlike reflection. We also scale everything according to the refractive indices keeping in mind that the water actually appears shallow when looked from above. This is stored in a 2D refraction texture which is then applied to the water surface by projective texturing.

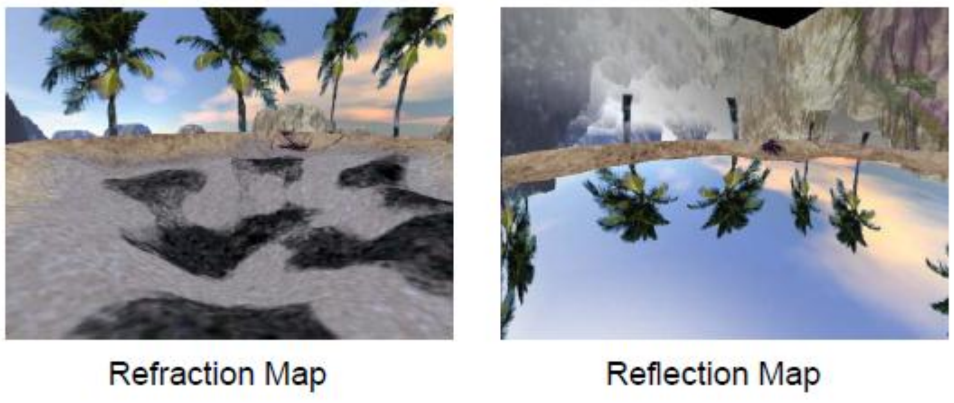


Figure 6: Reflection and Refraction maps

1.1.4. Fresnel term and rendering other various phenomena

Fresnel equation mentioned below is used to determine how much light gets reflected and what portion of it gets refracted.

But computing this for each point on water can be costly especially for large surfaces especially if real-time simulation is a requirement. To get past this problem, what we do is keep calculated values of the Fresnel term for different angles and store it in a 1D texture as a lookup table (See reference 9).

SECTION 2: MY IMPLEMENTATION

This implementation was done on my Dell laptop equipped with an Intel Core 2 Duo processor and a display adapter of the ATI Mobility Radeon HD 4500 Series (Microsoft Corporation WDDM 1.1).
I basically implemented two different scenes. One is that of a lake which was implemented using C++ and OpenGL. The second scene is that of a larger water body. For the second scene I chose to use the projected grid concept as my baseline to start with because of it being very realistic. I implemented this using C++ and DirectX.

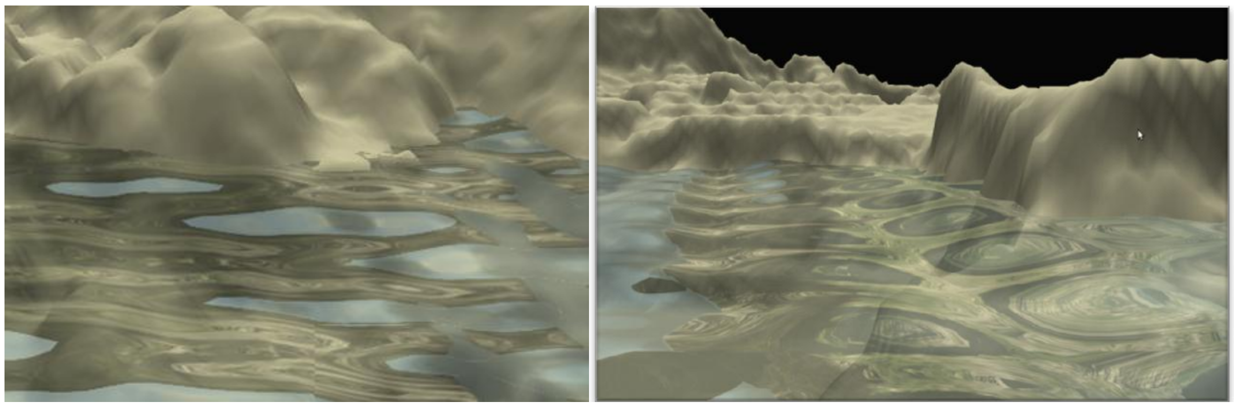
1st Implementation

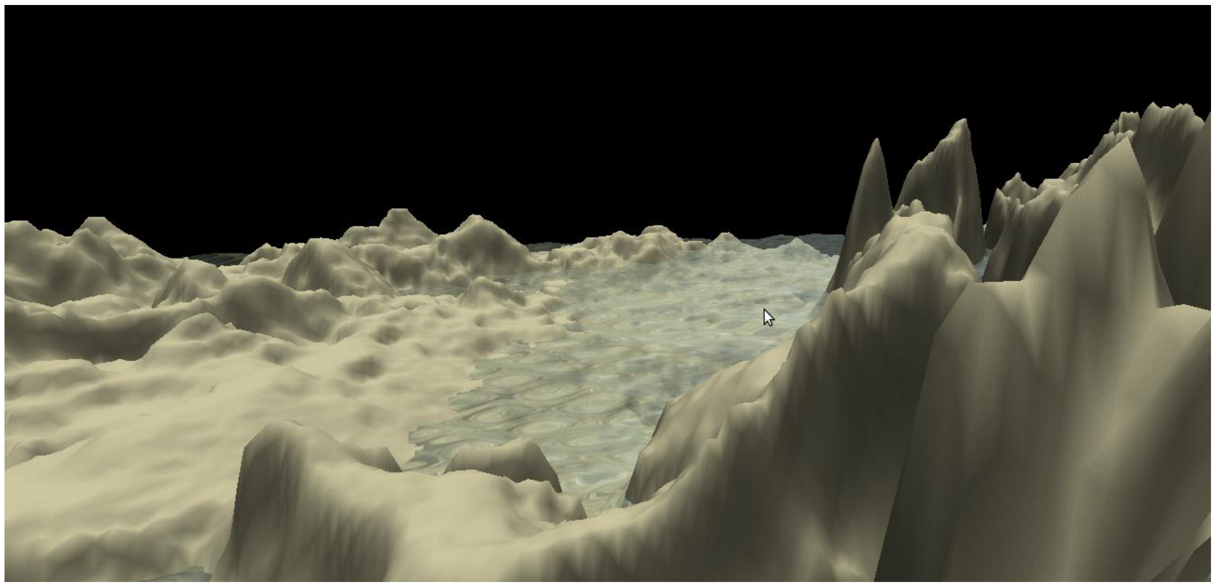
In this I create a very simplistic scene which might remind one of the games in the old days. The scene is that of a lake. This was implemented in C++ and OpenGL. The shader for water was written using GLSL. I chose to use a simple rectangular grid to represent my surface and stored the height map in a 2D texture. I intended to create smaller waves near the coast for more realism and therefore I stored the depth of water at every point for every vertex of the grid. But due to paying more attention to the second scene I couldn't accomplish this part. But the code can be easily extended to accomplish that. I have used a vertex shader to draw the mesh. This shader essentially sums up sine waves to get the surface. The tutorial for writing shaders in GLSL from http://nehe.gamedev.net/article/glsl_an_introduction/25007/ came in handy.

GLSL code for Water Shader:

```
uniform float t;                // Time
uniform int Waves;          // Number of waves
uniform float lambda[8];        // Wavelength
uniform float wHeight;          // Height of waves
uniform float WaveSp[8];        // Speed of waves
uniform float amp[8];           // Amplitude of waves
uniform vec2 dir[8];            // Direction of propagation
varying vec3 wNorm;         // Normal in world space
varying vec3 eNorm;         // Normal in eye space
varying vec3 pos;               // Position under consideration
const float pi = 3.14159;

float getFreq(int i) {
    float freq = 2*pi/lambda[i];
    return freq;
}

float getAngle(int i, float x, float y) {
    float ang = dotproduct(dir[i], vec2(x, y));
    return ang;
}

float getPhase(int i) {
    float phase = WaveSp[i] * getFreq(i);
    return phase;
}

float wave(int i, float x, float y) {
    return amp[i] * sin(getAngle(i,x,y) * getFreq(i) + t * getPhase(i));
}

float waveAmp(float x, float y) {
    float h = 0.0;
    for (int i = 0; i < Waves; ++i)
        h += wave(i, x, y);
    return h;
}


/* Now that we have represented Amplitude of waves as a function. We can calculate the /* normal at every point very easily by using the functions below.*/


float dxWave(int i, float x, float y) {
    float freq = getFreq(i);
    float A = amp[i] * dir[i].x * freq;
    return A * cos(getAngle(i,x,y) * freq + t * getPhase(i));
}

float dyWave(int i, float x, float y) {
    float freq = getFreq(i);
    float A = amp[i] * dir[i].y * freq;
    return A * cos(getAngle(i,x,y) * freq + t * getPhase(i));
}

vec3 getNormal(float x, float y) {
    float derivativeX = 0.0;
    float derivativeY = 0.0;
    for (int i = 0; i < Waves; ++i) {
        derivativeX += dxWave(i,x,y);
        derivativeY += dyWave(i,x,y);
    }
    vec3 norm = vec3(-derivativeX, -derivativeY, 1.0);
    return normalize(norm);
}
```

Results





Transparency above is set to be at 0.5 and the environmental reflections have been done with a cube map since we don't have any moving objects other than water itself. The height of the water at each point of the grid is a function of the horizontal coordinates as well as time. We can control the number of waves, their height, their wavelength, the speed as well as the direction in which each wave is moving.

2nd Implementation

This implementation is that of a much larger water body than the previous one. I chose to base this on the projective grid concept introduced by Claes Johanson in his PhD Thesis due to its high realism.

I first ported the code from Direct3D 9 to Direct3D 10. Doing this helped a lot in understanding the nuances of the projected grid algorithm. This method uses Perlin Noise to generate the waves but I have also incorporated the FFT Technique to generate the height maps for the waves as introduced by Tessendorf (See reference 11). I have also incorporated the real time realistic ocean lighting introduced by Bruneton et al (See reference 10) into my ocean scene. Deferred shading was done to speed up the process. Sound effects were also added for when we are above or below water. A shader for water below the surface was also added. Projector elevation is now a configurable parameter which now adjusts itself according to the height of the camera unless its set to manual. Below are the main shaders that I implemented for the above purposes.

HLSL Shader for Water surface rendering

```
//=============================================================================
// File: Water.fx
//
// Desc: Effect file for rendering projected grid sea water. Modified version of
//the technique developed by Claes Johanson.
//=============================================================================

//-----------------------------------------------------------------------------
// Global variables
//-----------------------------------------------------------------------------

cbuffer cbConstant
{
    float       LODbias;
    float       sun_shininess, sun_strength;
    float3      sun_vec;
    float       reflrefr_offset;

}

cbuffer cbDynamic
{
    float4x4    mViewProj;
    float4x4    mView;
    float4      view_position;
    float       g_fTime;
    float3      g_CameraPos;
}
```

```
TextureCube EnvironmentMap;
Texture2D FresnelMap;
Texture2D Normalmap;
Texture2D Refractionmap;
Texture2D Reflectionmap;

struct VS_INPUT
{
    float3 Pos      : POSITION;
    float3 Normal   : NORMAL;
    float2 tc       : TEXCOORD0;
};

struct VS_OUTPUT
{
    float4  Pos         : SV_POSITION;
    float2  tc          : TEXCOORD0;
    float3  normal      : TEXCOORD1;
    float3  viewvec     : TEXCOORD2;
    float3  screenPos   : TEXCOORD3;
    float3  worldPos    : TEXCOORD4;
    float3  screenPosDis    : TEXCOORD5;
};

SamplerState fresnel
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU  = CLAMP;
    AddressV  = CLAMP;
};

SamplerState nmap
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU  = CLAMP;
    AddressV  = CLAMP;
};

SamplerState refrmap
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU  = CLAMP;
    AddressV  = CLAMP;

};

SamplerState reflmap
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU  = CLAMP;
    AddressV  = CLAMP;

};

float3 water_scattering(float3 InColor, float depth, float length)
{
    float extinction = exp(-0.02f*max(depth,0));
    float alpha = (1 - extinction*exp(-max(length,0)*0.2f));
    return lerp(InColor, 0.7f*float3(0.157, 0.431, 0.706), alpha);
}

//---------------------------------------------------------------------------
// Desc: Transform local position, texture coordinate and output them to
//      pixel shader
//---------------------------------------------------------------------------

VS_OUTPUT VS(VS_INPUT i)
{
    VS_OUTPUT   o;
    o.worldPos = i.Pos.xyz;

    o.Pos = mul(float4(i.Pos.xyz,1), mViewProj);
    o.normal = normalize(i.Normal.xyz);
    o.viewvec = normalize(i.Pos.xyz - view_position.xyz);
    o.tc = i.tc;

    o.screenPosDis = (o.Pos.xyz / o.Pos.w);
    o.screenPosDis.xy = 0.5 + 0.5*o.screenPosDis.xy;//*float2(1,-1);
    o.screenPosDis.z = reflrefr_offset/o.screenPosDis.z;

    // alternate screenpos

    float4 tpos = mul(float4(i.Pos.x,0,i.Pos.z,1), mViewProj);
    o.screenPos = tpos.xyz/tpos.w;
    o.screenPos.xy = 0.5 + 0.5*o.screenPos.xy*float2(1,-1);
    o.screenPos.z = reflrefr_offset/o.screenPos.z;

    return o;
}

float4 PS(VS_OUTPUT i) : SV_Target
{
    float4 ut;
    ut.a = 1;
    float3 v = i.viewvec;
    float3 N = 2*Normalmap.Sample(nmap,i.tc)-1;
    float3 R = normalize(reflect(v,N));
    float f = FresnelMap.Sample(fresnel,dot(R,N));
    float3 light_dir = normalize(-sun_vec);

    // Calculate the inscatter color
    float3 projectedPoint = i.posW * 0.001f + earthPos;
    float3 projectedCamera = eye_pos * 0.001f + earthPos;
    float3 seaColor = float3(0.17, 0.27, 0.26);

    float3 scatterColor = groundColorFlat(projectedPoint, projectedCamera,
light_dir, float3(0, 0, 1), seaColor);
    scatterColor = hdr(scatterColor);

    // compute the reflection from sunlight
    float3 sunlight = sun_strength*pow(saturate(dot(R,
light_dir)),sun_shininess)*scatterColor;

    // Specular and reflection
    float4 refl = Reflectionmap.SampleBias(reflmap,i.screenPos.xy
i.screenPos.z*N.xz,LODbias);
    float3 skyrefl = EnvironmentMap.SampleBias(sky, R, LODbias);
    float3 col = lerp(sunlight+scatterColor+skyrefl,refl.rgb,refl.a);

    // Refraction and water color
    float3 refr = Refractionmap.Sample(refrmap,i.screenPos.xy-
i.screenPos.z*N.xz);

    // mix reflection/specular with refraction/color
    ut.rgb = lerp( refr, col, f );


    return ut;
}

technique10 water
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_4_0, VS() ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, PS() ) );
    }
}
```

HLSL Shader for Generating Height maps Using Perlin Noise

```
//=============================================================================
//File: WaterHeightMapGen.fx
//Desc: Effect file for rendering Height Maps using Perlin Noise.
//=============================================================================

struct VS_INPUT
{
    float3 Pos       : POSITION;
    float3 Normal    : NORMAL;
    float2 tc        : TEXCOORD0;
};

struct VS_OUTPUT
{
      float4  Pos               : SV_POSITION;
      float2  tc0               : TEXCOORD0;
      float2  tc1               : TEXCOORD1;
};

struct PS_OUTPUT
{
    float4 Color : COLOR;
};


//-----------------------------------------------------------------------------
// Global variables
//-----------------------------------------------------------------------------
float       scale;

Texture2D   noise0;
Texture2D   noise1;
```

```
// samplers
SamplerState N0
{
      Filter = MIN_MAG_MIP_LINEAR;
      AddressU  = WRAP;
      AddressV  = WRAP;
};

SamplerState N1
{
      Filter = MIN_MAG_MIP_LINEAR;
      AddressU  = WRAP;
      AddressV  = WRAP;
};


VS_OUTPUT VS(VS_INPUT i)
{
      VS_OUTPUT o;
      o.Pos = float4( i.tc.x*2-1,1-i.tc.y*2, 0, 1 );
      o.tc0 = scale*i.Pos.xz*0.007813;
      o.tc1 = scale*i.Pos.xz*0.125;
      return o;
}


float4 PS(VS_OUTPUT i) : SV_Target
{
    float c = noise0.Sample(N0, i.tc0) + noise1.Sample(N1, i.tc1) - 0.5;
    return float4( c, c, c, 1 );
}

technique10 T0
{
      pass P0
      {
        SetVertexShader( CompileShader( vs_4_0, VS() ) );
        SetPixelShader( CompileShader( ps_4_0, PS() ) );

        CullMode = NONE;
      }
}
```

HLSL Shader for Generating Height maps Using FFT

```
//===========================================================================
//File: WaterHeightMapGen2.fx
//Desc: Effect file for generating the height map using FFT.
// This technique was developed by Tessendorf.
//===========================================================================

Texture2D ButterflyTex;
Texture2D SourceTex0;
Texture2D SourceTex1;
Texture2D SourceTex2;

cbuffer cbConstant
{
      float Dimension;
}

cbuffer cbDynamic
{
      float ButterflyIndex;
      bool IsLastPass;
}


SamplerState pointSampler
{
    Filter = MIN_MAG_MIP_POINT;
    AddressU = Clamp;
    AddressV = Clamp;
};

SamplerState linearSampler
{
    Filter = MIN_MAG_MIP_POINT;
    AddressU = Mirror;
    AddressV = Mirror;
};


void FFTVS( float4 vInPos            : POSITION,
            float2 vInTex            : TEXCOORD,
            out float2 vOutTex       : TEXCOORD0,
            out float4 vOutPos       : SV_POSITION)
{
      vOutTex = vInTex;
      vOutPos = vInPos;
}

//==========================================================================
//Desc: Performs an horizontal FFT on the nth row with an in-place height computation
//==========================================================================

void FFTHorizontalPS(float2 vInTex                : TEXCOORD0,
                     out float4 vOutColor0        : SV_TARGET0,
                     out float4 vOutColor1        : SV_TARGET1)
{
      float4 vIndexAndWeight = ButterflyTex.Sample(pointSampler,
      float2(vInTex.x, ButterflyIndex));
      float2 vIndex = vIndexAndWeight.rg;
      float2 vWeight = vIndexAndWeight.ba;

      float2 vSourceA = SourceTex0.Sample(linearSampler, float2(vIndex.x,
vInTex.y)).rg;
      float2 vSourceB = SourceTex0.Sample(linearSampler, float2(vIndex.y,
vInTex.y)).rg;

      float2 vWeightedB;
      vWeightedB.r = vWeight.r * vSourceB.r - vWeight.g * vSourceB.g;
      vWeightedB.g = vWeight.g * vSourceB.r + vWeight.r * vSourceB.g;

      float2 vComplex = vSourceA + vWeightedB;

      vSourceA = SourceTex1.Sample(linearSampler, float2(vIndex.x,
vInTex.y)).rg;
      vSourceB = SourceTex1.Sample(linearSampler, float2(vIndex.y,
vInTex.y)).rg;

      vWeightedB.r = vWeight.r * vSourceB.r - vWeight.g * vSourceB.g;
      vWeightedB.g = vWeight.g * vSourceB.r + vWeight.r * vSourceB.g;

      float2 vComplex0 = vSourceA + vWeightedB;

      vSourceA = SourceTex1.Sample(linearSampler, float2(vIndex.x,
vInTex.y)).ba;
      vSourceB = SourceTex1.Sample(linearSampler, float2(vIndex.y,
vInTex.y)).ba;

      vWeightedB.r = vWeight.r * vSourceB.r - vWeight.g * vSourceB.g;
      vWeightedB.g = vWeight.g * vSourceB.r + vWeight.r * vSourceB.g;

      float2 vComplex1 = vSourceA + vWeightedB;

      vOutColor0 = float4(vComplex, 0, 1);
      vOutColor1 = float4(vComplex0, vComplex1);
}

//==========================================================================
//Desc: Performs an vertical FFT on the nth column with an in-place height computation
//==========================================================================

void FFTVerticalPS(     float2 vInTex            : TEXCOORD0,
                        out float4 vOutColor0    : SV_TARGET0,
                        out float4 vOutColor1    : SV_TARGET1)
{
      float4 vIndexAndWeight = ButterflyTex.Sample(pointSampler,
float2(vInTex.y, ButterflyIndex));
      float2 vIndex = vIndexAndWeight.rg;
      float2 vWeight = vIndexAndWeight.ba;

      float2 vSourceA = SourceTex0.Sample(linearSampler, float2(vInTex.x,
vIndex.x)).rg;
      float2 vSourceB = SourceTex0.Sample(linearSampler, float2(vInTex.x,
vIndex.y)).rg;

      float2 vWeightedB;
      vWeightedB.r = vWeight.r * vSourceB.r - vWeight.g * vSourceB.g;
      vWeightedB.g = vWeight.g * vSourceB.r + vWeight.r * vSourceB.g;

      float2 vComplex = vSourceA + vWeightedB;

      vSourceA = SourceTex1.Sample(linearSampler, float2(vInTex.x,
vIndex.x)).rg;
      vSourceB = SourceTex1.Sample(linearSampler, float2(vInTex.x,
vIndex.y)).rg;

      vWeightedB.r = vWeight.r * vSourceB.r - vWeight.g * vSourceB.g;
      vWeightedB.g = vWeight.g * vSourceB.r + vWeight.r * vSourceB.g;

      float2 vComplex0 = vSourceA + vWeightedB;

      vSourceA = SourceTex1.Sample(linearSampler, float2(vInTex.x,
vIndex.x)).ba;
```

```
	vSourceB = SourceTex1.Sample(linearSampler, float2(vInTex.x, 
vIndex.y)).ba;

	vWeightedB.r = vWeight.r * vSourceB.r - vWeight.g * vSourceB.g;
	vWeightedB.g = vWeight.g * vSourceB.r + vWeight.r * vSourceB.g;

	float2 vComplex1 = vSourceA + vWeightedB;


	if(!IsLastPass)
	{
		vOutColor0 = float4(vComplex, 0.0f, 1.0f);
		vOutColor1 = float4(vComplex0, vComplex1);
	}
	else
	{
		if((vInTex.x * Dimension + vInTex.y * Dimension) % 2 == 1)
		{
			vComplex.r = -vComplex.r;
			vComplex0.r = -vComplex0.r;
			vComplex1.r = -vComplex1.r;
		}
		vOutColor0 = float4(vComplex0.r, vComplex.r, vComplex1.r, 0);
	}
}

technique11 HorizontalFFT
{
	pass P0
	{
	  SetVertexShader(CompileShader(vs_4_0, FFTVS()));
	  SetPixelShader(CompileShader(ps_4_0, FFTHorizontalPS()));

	}
}

technique11 VerticalFFT
{
	pass P0
	{
	  SetVertexShader(CompileShader(vs_4_0, FFTVS()));
	  SetPixelShader(CompileShader(ps_4_0,  FFTVerticalPS()));

	}
}
```

HLSL Shader for Generating Normal Maps

```
//=============================================================================
// File: WaterNormalMapGen.fx
//
// Desc: Effect file for generating Normal maps. Derived mostly from the shader provided by Claes
//=============================================================================


float inv_mapsize_x,inv_mapsize_y;
float4	corner00, corner01, corner10, corner11;
float amplitude;
```

```
struct VS_INPUT
{
    float3 Pos      : POSITION;
    float3 Normal   : NORMAL;
    float2 tc       : TEXCOORD0;
};

struct VS_OUTPUT
{
      float4  Pos              : SV_POSITION;
      float2  tc               : TEXCOORD0;
      float3  tc_p_dx          : TEXCOORD1;
      float3  tc_p_dy          : TEXCOORD2;
      float3  tc_m_dx          : TEXCOORD3;
      float3  tc_m_dy          : TEXCOORD4;
};

Texture2D hmap;

// samplers
SamplerState hsampler
{
      AddressU  = WRAP;
      AddressV  = WRAP;
      Filter = MIN_MAG_MIP_LINEAR;
};


VS_OUTPUT VS(VS_INPUT i)
{
      VS_OUTPUT o;
      o.Pos = float4( i.tc.x*2-1,1-i.tc.y*2, 0, 1 );
      float scale = 1;
      float2 tc = i.tc + float2(-inv_mapsize_x*scale,0);
      float4 meh =
lerp(lerp(corner00,corner01,tc.x),lerp(corner10,corner11,tc.x),tc.y);
      o.tc_m_dx = meh.xyz/meh.w;

      tc = i.tc + float2(+inv_mapsize_x*scale,0);
      meh =
lerp(lerp(corner00,corner01,tc.x),lerp(corner10,corner11,tc.x),tc.y);
      o.tc_p_dx = meh.xyz/meh.w;

      tc = i.tc + float2(0,-inv_mapsize_y*scale);
      meh =
lerp(lerp(corner00,corner01,tc.x),lerp(corner10,corner11,tc.x),tc.y);
      o.tc_m_dy = meh.xyz/meh.w;

      tc = i.tc + float2(0,inv_mapsize_y*scale);
      meh =
lerp(lerp(corner00,corner01,tc.x),lerp(corner10,corner11,tc.x),tc.y);
      o.tc_p_dy = meh.xyz/meh.w;

      o.tc = i.tc;

      return o;
}
```

```
float4 PS(VS_OUTPUT i) : SV_Target
{
      float2      dx = {inv_mapsize_x,0},
                  dy = {0,inv_mapsize_y};
      i.tc_p_dx.y = amplitude*hmap.Sample(hsampler, i.tc+dx);
      i.tc_m_dx.y = amplitude*hmap.Sample(hsampler, i.tc-dx);
      i.tc_p_dy.y = amplitude*hmap.Sample(hsampler, i.tc+dy);
      i.tc_m_dy.y = amplitude*hmap.Sample(hsampler, i.tc-dy);
      float3   normal   =   normalize(-cross(i.tc_p_dx-i.tc_m_dx,   i.tc_p_dy-
i.tc_m_dy));
      return float4(0.5+0.5*normal,1);
}

technique10 T0
{
      pass P0
      {
        SetVertexShader( CompileShader( vs_4_0, VS() ) );
        SetPixelShader( CompileShader( ps_4_0, PS() ) );
      }
}
```

HLSL Shader for Skybox

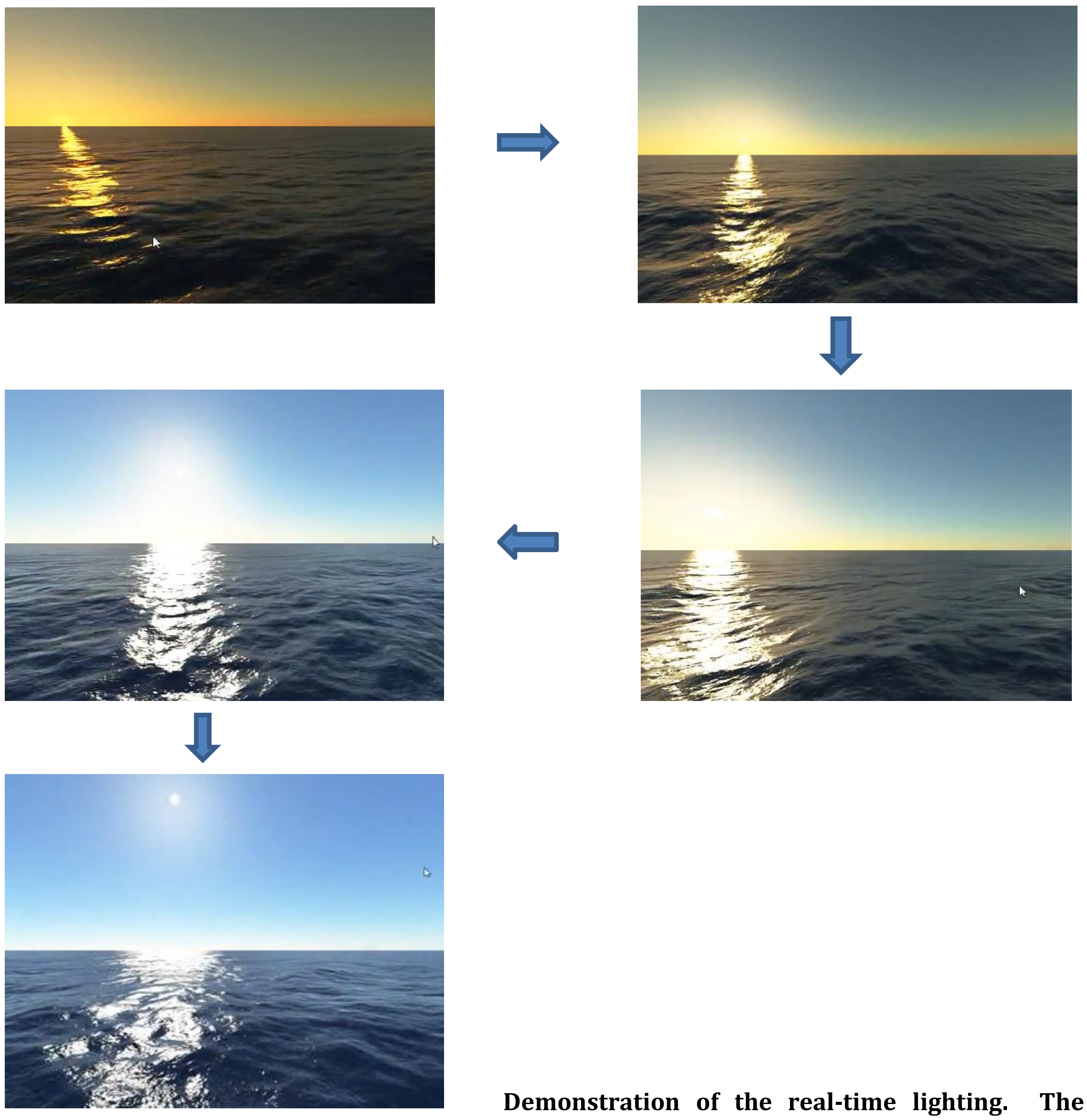
```
//==============================================================================
// File: Skybox.fx
//
// Desc: Effect file for rendering skybox
//==============================================================================

//--------------------------------------------------------------------------------------
// Global variables
//--------------------------------------------------------------------------------------

cbuffer cb0
{
      float4x4 mViewProj;      // View * Projection matrix
      float sunAlpha, sunTheta, sunShininess, sunStrength; // variables for Skybox sun
      bool underwater;  // set if camera is underwater
}

TextureCube EnvironmentMap;    // Sky texture

//--------------------------------------------------------------------------------------
// SamplerStates
//--------------------------------------------------------------------------------------
SamplerState samSkybox
{
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU  = CLAMP;
    AddressV  = CLAMP;
    AddressW  = CLAMP;
};

//--------------------------------------------------------------------------------------
// Input/Output Structures
```

```
//--------------------------------------------------------------------------------------


struct RenderSkyboxVSOUT
{
      float4  Position    : SV_POSITION0; // position of sphere
      float3  SunPosition      : TEXCOORD0;       // position of sun
      float3      SkyTexUVW    : TEXCOORD1;       // texture coordinates
      float WorldYPos    : TEXCOORD2;       // Y position of vertex
};

//--------------------------------------------------------------------------------------
// Shader Functions
//--------------------------------------------------------------------------------------

RenderSkyboxVSOUT RenderSkyboxVS( float4 vPos : POSITION )
{
      RenderSkyboxVSOUT Output;
      float zfar = 4000;

      Output.WorldYPos = saturate(vPos.y);

      Output.Position = mul( zfar*vPos, mViewProj );

      Output.SkyTexUVW = vPos;
      Output.SunPosition.x = cos(sunTheta)*sin(sunAlpha);
      Output.SunPosition.y = sin(sunTheta);
      Output.SunPosition.z = cos(sunTheta)*cos(sunAlpha);

      return Output;
}

float4 RenderSkyboxPS( RenderSkyboxVSOUT Input ) : SV_TARGET
{
      if(underwater)
            return float4( .02f, .025f, .035f, 1.0f );

      float4 Output;

    float3 sunlight = sunStrength*pow(
saturate(dot(normalize(Input.SkyTexUVW), Input.SunPosition)),
            sunShininess)*float3(1.2, 0.4, 0.1);

      Output.a = 1;

      Output.rgb  =  (g_EnvironmentMap.Sample(  samSkybox,Input.SkyTexUVW  )+
sunlight);
      return Output;
}

//--------------------------------------------------------------------------------------
// Shader Techniques
//--------------------------------------------------------------------------------------

technique10 RenderSkybox
{
      pass P0
      {
            SetVertexShader( CompileShader( vs_4_0, RenderSkyboxVS() ) );
            SetPixelShader( CompileShader( ps_4_0, RenderSkyboxPS() ) );
    }
}
```

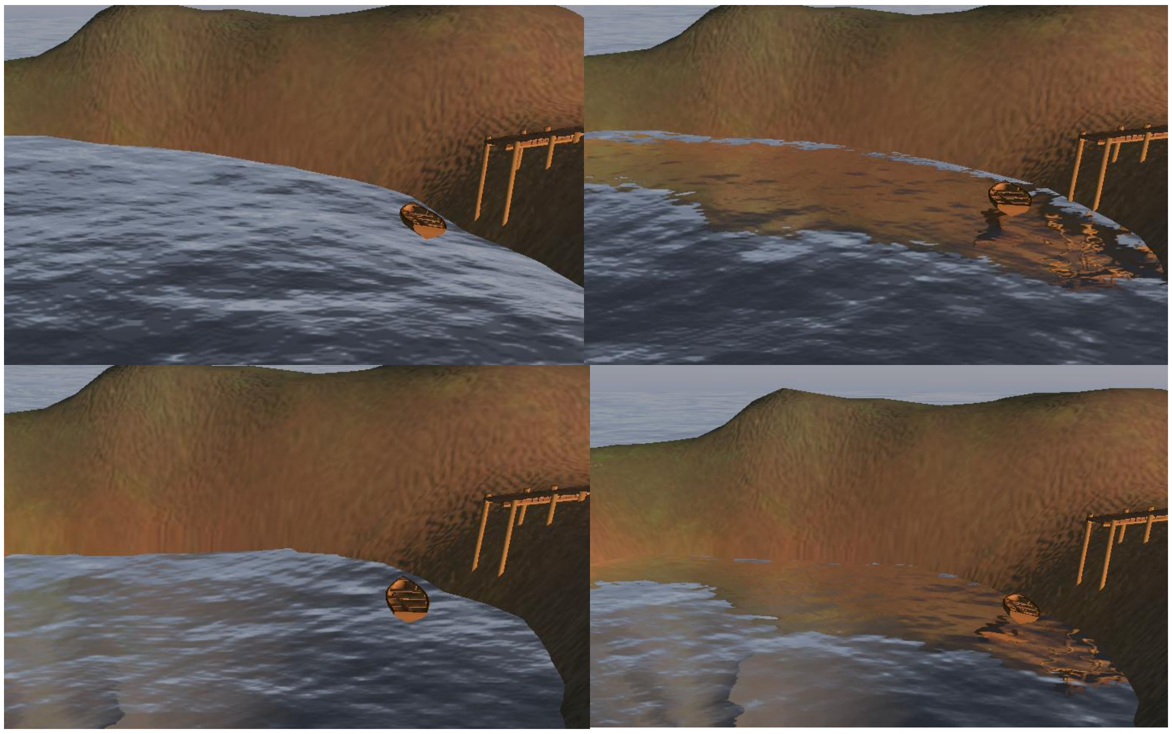
HLSL Shader for water under the surface

```
//==========================================================================
// File: underwater.fx
//
// Desc: Effect file for rendering water under the surface
//==========================================================================
TextureCube EnvironmentMap;
Texture2D   FresnelMap;
Texture2D   Normalmap;

cbuffer cbConstant
{
    float         LODbias;
    float         sun_alfa, sun_theta, sun_shininess, sun_strength;
    float3        watercolour;
}

cbuffer cbDynamic
{
    float4x4      mViewProj;
    float4        view_position;
    float         g_fTime;
    bool          IsUnderwater;
}

struct VS_INPUT
{
      float3 Pos      : POSITION;
      float3 Normal   : NORMAL;
      float2 tc       : TEXCOORD0;
};

struct VS_OUTPUT
{
      float4  Pos       : SV_POSITION;
      float2  tc        : TEXCOORD0;
      float3  normal    : TEXCOORD1;
      float3  viewvec   : TEXCOORD2;
      float3  sun       : TEXCOORD3;
      float3  worldPos  : POSITION0;
};

SamplerState sky
{
      Filter = MIN_MAG_MIP_LINEAR;
      AddressU  = WRAP;
      AddressV  = WRAP;
      AddressW  = WRAP;
};

SamplerState fresnel
{
      Filter = MIN_MAG_MIP_LINEAR;
      AddressU  = CLAMP;
      AddressV  = CLAMP;
};

SamplerState nmap
{
      Filter = MIN_MAG_MIP_LINEAR;
      AddressU  = CLAMP;
      AddressV  = CLAMP;
};

float4 CalcWaterCoeff( float3 WorldPos )
{
      float CameraDist;
    CameraDist = distance( view_position.xyz, WorldPos );

      return  exp(-(0.1f*CameraDist));
}

VS_OUTPUT VS(VS_INPUT i)
{
    VS_OUTPUT   o;
    o.worldPos = i.Pos.xyz/4;

    o.Pos = mul(float4(i.Pos.xyz,1), mViewProj);
      o.normal = normalize(i.Normal.xyz);
      o.viewvec = i.Pos.xyz - view_position.xyz/view_position.w;
      o.tc = i.tc;
      o.sun.x = cos(sun_theta)*sin(sun_alfa);
      o.sun.y = sin(sun_theta);
      o.sun.z = cos(sun_theta)*cos(sun_alfa);
      return o;
}

float4 PS(VS_OUTPUT i) : SV_Target
{
    float4 ut;

      ut.a = 1;
      float3 v = normalize(i.viewvec);
      float3 N = 2*Normalmap.Sample(nmap,i.tc)-1;
      //float3 N = float3(1, 1, 1);
      float3 R = refract(v,N,1.33);
      R.y = - R.y;
      float f = FresnelMap.Sample(fresnel,dot(R,N));
      float sunlight = sun_strength*pow(saturate(dot(R,
i.sun)),sun_shininess);
    float3 col = EnvironmentMap.SampleBias(sky, R, LODbias) +
sunlight*float3(1.2, 0.7, 0.3);
      float3 reflcol = watercolour + saturate(30*float3(0.4,0.6,0.8)*(1-
dot(float3(0,1,0),N)));
      ut.rgb = lerp(col, reflcol, f) ;

      if( IsUnderwater )
      {
            float4 vWaterColor = { .02f, .025f, .035f, 1.0f };
            float f = CalcWaterCoeff(i.worldPos);
            return (ut*f + vWaterColor*(1-f));
      }
    return ut;}
technique10 underwater
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_4_0, VS() ) );
        SetPixelShader( CompileShader( ps_4_0, PS() ) );
      }
}
```

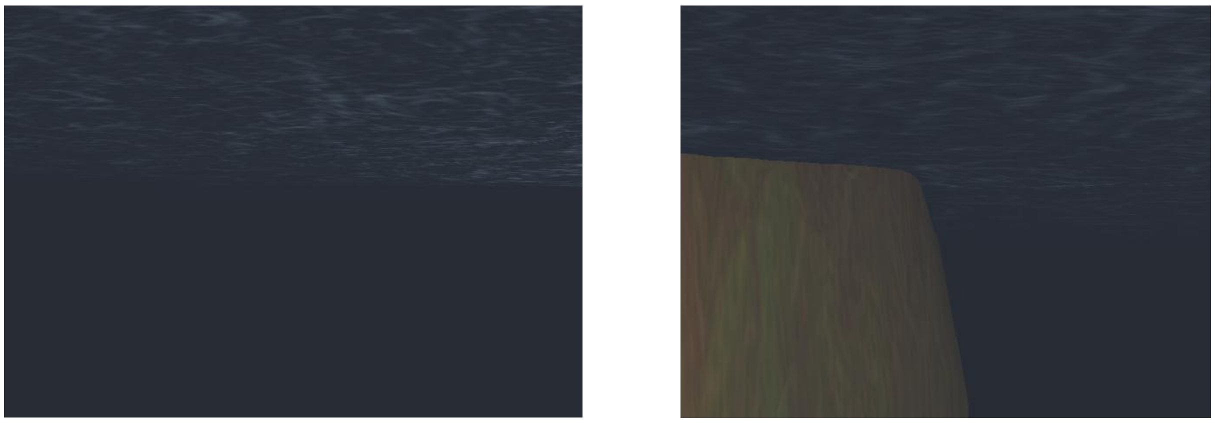
Results



Demonstration of the real-time lighting. The scattering color due to the sun was blended with the terrain texture so as to create appropriate sky color according to the time of the day. The water surface is being generated using the FFT.



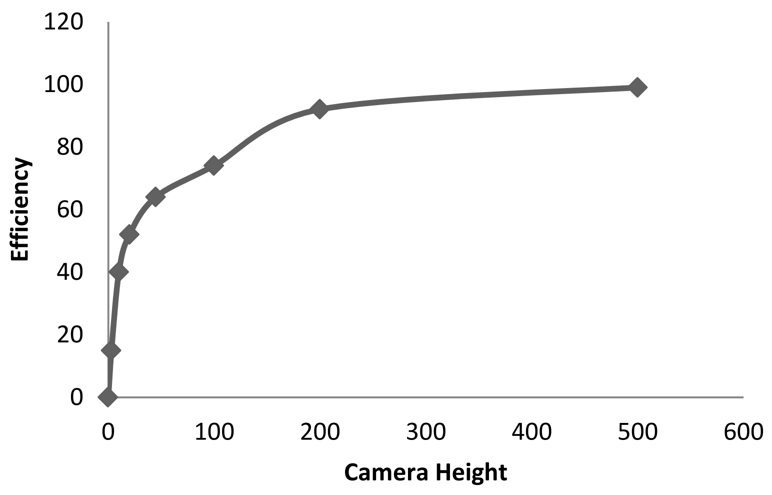
Reflection and Refraction. The surface is being generated using Perlin Noise.
Top left) Neither reflection nor refraction is enabled.
Top right) Only reflection has been enabled.
Bottom left) Only refraction has been enabled.
Bottom right) Both refraction and reflection are enabled.



Underwater

Observations

This algorithm usually renders with 0.6-0.9 efficiency (which can be defined as ratio of rendered volume to visible volume). FFT water is more computationally heavy than the perlin noise water by almost 15-35 fps depending on the various parameters. The efficiency drops drastically when the camera is very close to the water surface almost to 0.05. It however tends to increase suddenly when we move up before saturating near almost 1 at very great heights. Efficiency seems to vary almost logarithmically with the height of the camera.



Known Artifacts

1. This method doesn't work fine at high altitudes as several artifacts start appearing. The land starts appearing way too wavy.

2. Changing the camera angle too quick at times leads to an artifact on the edges of the screen which immediately disappears.

Possible Future Work

1. Use of two or more heightmaps to generate interesting waves in 3D.
2. Use floating point depth buffer so as to get rid of the artifact due to camera height as explained on http://outerra.blogspot.in/search/label/depth%20buffer

Image Credits

Figure 1) Real Time Fluids in Games, Matthias Muller-Fischer, Siggraph2006
Figure 2) http://geoinformatics.fsv.cvut.cz/wiki/images
Figure 3,4) Real-time water rendering: Introducing the projected grid concept, Claes Johanson, Master Thesis, 2004
Figure 5) http://libnoise.sourceforge.net/tutorials/tutorial4.html#octaves
Figure 6) Z. Jeremy, "Vertex Texture Fetch Water" 2004 (NVIDIA).

References

1. Kei Iwasaki, Yoshinori Dobashi and Tomoyuki Nishita - An Efficient Method for Rendering Underwater Optical Effects Using Graphics Hardware – COMPUTER GRAPHICS FORUM, 2002.

2. Morgan Kaufmann: Level of Detail for 3D Graphics

3. Real-time water rendering: Introducing the projected grid concept, Claes Johanson, Master Thesis, 2004

4. Jeremy, “Vertex Texture Fetch Water” 2004 (NVIDIA).

5. http://libnoise.sourceforge.net/noisegen/index.html

6. http://freespace.virgin.net/hugo.elias/models/m_perlin.htm

7. http://people.csail.mit.edu/acornejo/Projects/html/hmap.htm

8. http://www.modwiki.net/wiki/Cube_maps

9. http://habib.wikidot.com/techniques

10. “Real-time Realistic Ocean Lighting using Seamless Transitions from Geometry to BRDF” by Eric Bruneton and Fabrice Neyret and Nicolas Holzschuch (2009)

11. Tessendorf, Jerry. Simulating Ocean Water. In SIGGRAPH 2002 Course Notes #9 (Simulating Nature: Realistic and Interactive Techniques), ACM Press.