DirectX 11 Tutorial 17 Part2 – Water and Normal Mapping

This week's tutorial focuses on Ocean Vertex and Pixel Shaders that have been modified from NVIDIA's Ocean.fx effect file (the original file is listed at the end).

You are required to:

Scene.cpp

1. Load the ocean vertex and pixel shaders
2. Load the water normal map (“waves.dds”)
3. Create a water object of class Grid provide the environment texture and the normal map to the Grid constructor

Ocean Vertex Shader

1. Modify the frequency of the wave displacement

Ocean Pixel Shader

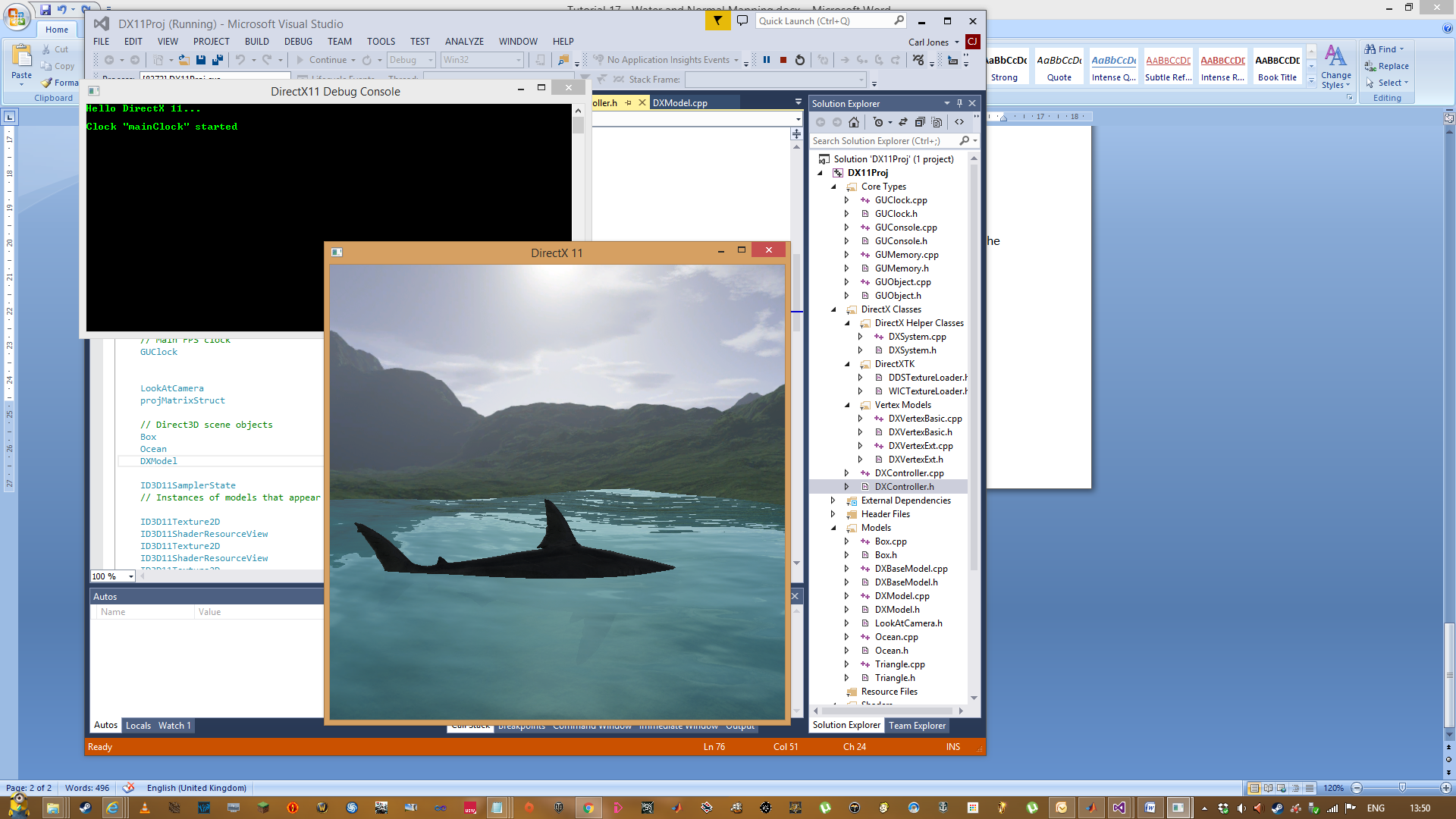
1. Modify the Fresnel parameters and observe the effect

Make minor changes to the Scene.cpp source

1. Modify the default blend states for the water Effect to enable alpha blending, this is not explicitly given in the lecture notes you will need to do some research.
2. Add a shark to the scene
3. modify the world transform matrices for the shark and to make it move (::updateScene())

Use the source code from your solution week 17

Review the week 17 lecture notes "Lecture 17 - Water and Normal Mapping” and complete the tasks listed above.



Screen shot of required outcome.

Listing of original ocean effect from Nvidia:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*NVMH3\*\*\*\*

Path: NVSDK\Common\media\cgfx

File: ocean.fx

Copyright NVIDIA Corporation 2003

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, THIS SOFTWARE IS PROVIDED

\*AS IS\* AND NVIDIA AND ITS SUPPLIERS DISCLAIM ALL WARRANTIES, EITHER EXPRESS

OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY

AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL NVIDIA OR ITS SUPPLIERS

BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES

WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS,

BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS)

ARISING OUT OF THE USE OF OR INABILITY TO USE THIS SOFTWARE, EVEN IF NVIDIA HAS

BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Comments:

Simple ocean shader with animated bump map and geometric waves

Based partly on "Effective Water Simulation From Physical Models", GPU Gems

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

float Script : STANDARDSGLOBAL <

string UIWidget = "none";

string ScriptClass = "object";

string ScriptOrder = "standard";

string ScriptOutput = "color";

string Script = "Technique=PS20;";

> = 0.8;

float4x4 worldMatrix : World < string UIWidget = "none";>; // World or Model matrix

float4x4 wvpMatrix : WorldViewProjection < string UIWidget = "none";>; // Model\*View\*Projection

float4x4 worldViewMatrix : WorldView < string UIWidget = "none";>;

float4x4 viewInverseMatrix : ViewInverse < string UIWidget = "none";>;

float time : Time < string UIWidget = "none"; >;

texture normalMap : Normal

<

string ResourceName = "waves2.dds";

string ResourceType = "2D";

>;

texture cubeMap : Environment

<

string ResourceName = "CloudyHillsCubemap2.dds";

string ResourceType = "Cube";

>;

sampler2D normalMapSampler = sampler\_state

{

Texture = <normalMap>;

#if 0

// this is a trick from Halo - use point sampling for sparkles

MagFilter = Linear;

MinFilter = Point;

MipFilter = None;

#else

MagFilter = Linear;

MinFilter = Linear;

MipFilter = Linear;

#endif

};

samplerCUBE envMapSampler = sampler\_state

{

Texture = <cubeMap>;

MinFilter = Linear;

MagFilter = Linear;

MipFilter = Linear;

AddressU = Clamp;

AddressV = Clamp;

};

float bumpHeight

<

string UIWidget = "slider";

float UIMin = 0.0; float UIMax = 2.0; float UIStep = 0.01;

string UIName = "Bump Height";

> = 0.1;

float2 textureScale

<

string UIName = "Texture scale";

> = { 8.0, 4.0 };

float2 bumpSpeed

<

string UIName = "Bumpmap translation speed";

> = { -0.05, 0.0 };

float fresnelBias

<

string UIName = "Fresnel bias";

string UIWidget = "slider";

float UIMin = 0.0; float UIMax = 1.0; float UIStep = 0.01;

> = 0.1;

float fresnelPower

<

string UIName = "Fresnel exponent";

string UIWidget = "slider";

float UIMin = 1.0; float UIMax = 10.0; float UIStep = 0.01;

> = 4.0;

float hdrMultiplier

<

string UIName = "HDR multiplier";

string UIWidget = "slider";

float UIMin = 0.0; float UIMax = 100.0; float UIStep = 0.01;

> = 3.0;

float4 deepColor : Diffuse

<

string UIName = "Deep water color";

> = {0.0f, 0.0f, 0.1f, 1.0f};

float4 shallowColor : Diffuse

<

string UIName = "Shallow water color";

> = {0.0f, 0.5f, 0.5f, 1.0f};

float4 reflectionColor : Specular

<

string UIName = "Reflection color";

> = {1.0f, 1.0f, 1.0f, 1.0f};

// these are redundant, but makes the ui easier:

float reflectionAmount

<

string UIName = "Reflection amount";

string UIWidget = "slider";

float UIMin = 0.0; float UIMax = 2.0; float UIStep = 0.01;

> = 1.0f;

float waterAmount

<

string UIName = "Water color amount";

string UIWidget = "slider";

float UIMin = 0.0; float UIMax = 2.0; float UIStep = 0.01;

> = 1.0f;

float waveAmp

<

string UIName = "Wave amplitude";

string UIWidget = "slider";

float UIMin = 0.0; float UIMax = 10.0; float UIStep = 0.1;

> = 1.0;

float waveFreq

<

string UIName = "Wave frequency";

string UIWidget = "slider";

float UIMin = 0.0; float UIMax = 1.0; float UIStep = 0.001;

> = 0.1;

struct a2v {

float4 Position : POSITION; // in object space

float2 TexCoord : TEXCOORD0;

float3 Tangent : TEXCOORD1;

float3 Binormal : TEXCOORD2;

float3 Normal : NORMAL;

};

struct v2f {

float4 Position : POSITION; // in clip space

float2 TexCoord : TEXCOORD0;

float3 TexCoord1 : TEXCOORD1; // first row of the 3x3 transform from tangent to cube space

float3 TexCoord2 : TEXCOORD2; // second row of the 3x3 transform from tangent to cube space

float3 TexCoord3 : TEXCOORD3; // third row of the 3x3 transform from tangent to cube space

float2 bumpCoord0 : TEXCOORD4;

float2 bumpCoord1 : TEXCOORD5;

float2 bumpCoord2 : TEXCOORD6;

float3 eyeVector : TEXCOORD7;

};

// wave functions

struct Wave {

float freq; // 2\*PI / wavelength

float amp; // amplitude

float phase; // speed \* 2\*PI / wavelength

float2 dir;

};

#define NWAVES 2

Wave wave[NWAVES] = {

{ 1.0, 1.0, 0.5, float2(-1, 0) },

{ 2.0, 0.5, 1.3, float2(-0.7, 0.7) }

};

float evaluateWave(Wave w, float2 pos, float t)

{

return w.amp \* sin( dot(w.dir, pos)\*w.freq + t\*w.phase);

}

// derivative of wave function

float evaluateWaveDeriv(Wave w, float2 pos, float t)

{

return w.freq\*w.amp \* cos( dot(w.dir, pos)\*w.freq + t\*w.phase);

}

// sharp wave functions

float evaluateWaveSharp(Wave w, float2 pos, float t, float k)

{

return w.amp \* pow(sin( dot(w.dir, pos)\*w.freq + t\*w.phase)\* 0.5 + 0.5 , k);

}

float evaluateWaveDerivSharp(Wave w, float2 pos, float t, float k)

{

return k\*w.freq\*w.amp \* pow(sin( dot(w.dir, pos)\*w.freq + t\*w.phase)\* 0.5 + 0.5 , k - 1) \* cos( dot(w.dir, pos)\*w.freq + t\*w.phase);

}

v2f BumpReflectWaveVS(a2v IN,

uniform float4x4 WorldViewProj,

uniform float4x4 World,

uniform float4x4 ViewIT,

uniform float BumpScale,

uniform float2 textureScale,

uniform float2 bumpSpeed,

uniform float time,

uniform float waveFreq,

uniform float waveAmp

)

{

v2f OUT;

wave[0].freq = waveFreq;

wave[0].amp = waveAmp;

wave[1].freq = waveFreq\*2.0;

wave[1].amp = waveAmp\*0.5;

float4 P = IN.Position;

// sum waves

P.y = 0.0;

float ddx = 0.0, ddy = 0.0;

for(int i=0; i<NWAVES; i++) {

P.y += evaluateWave(wave[i], P.xz, time);

float deriv = evaluateWaveDeriv(wave[i], P.xz, time);

ddx += deriv \* wave[i].dir.x;

ddy += deriv \* wave[i].dir.y;

}

// compute tangent basis

float3 B = float3(1, ddx, 0);

float3 T = float3(0, ddy, 1);

float3 N = float3(-ddx, 1, -ddy);

OUT.Position = mul(P, WorldViewProj);

// pass texture coordinates for fetching the normal map

OUT.TexCoord.xy = IN.TexCoord\*textureScale;

time = fmod(time, 100.0);

OUT.bumpCoord0.xy = IN.TexCoord\*textureScale + time\*bumpSpeed;

OUT.bumpCoord1.xy = IN.TexCoord\*textureScale\*2.0 + time\*bumpSpeed\*4.0;

OUT.bumpCoord2.xy = IN.TexCoord\*textureScale\*4.0 + time\*bumpSpeed\*8.0;

// compute the 3x3 tranform from tangent space to object space

float3x3 objToTangentSpace;

// first rows are the tangent and binormal scaled by the bump scale

objToTangentSpace[0] = BumpScale \* normalize(T);

objToTangentSpace[1] = BumpScale \* normalize(B);

objToTangentSpace[2] = normalize(N);

OUT.TexCoord1.xyz = mul(objToTangentSpace, World[0].xyz);

OUT.TexCoord2.xyz = mul(objToTangentSpace, World[1].xyz);

OUT.TexCoord3.xyz = mul(objToTangentSpace, World[2].xyz);

// compute the eye vector (going from shaded point to eye) in cube space

float4 worldPos = mul(P, World);

OUT.eyeVector = ViewIT[3] - worldPos; // view inv. transpose contains eye position in world space in last row

return OUT;

}

// Pixel Shaders

float4 BumpReflectPS20(v2f IN,

uniform sampler2D NormalMap,

uniform samplerCUBE EnvironmentMap) : COLOR

{

// fetch the bump normal from the normal map

float4 N = tex2D(NormalMap, IN.TexCoord.xy)\*2.0 - 1.0;

float3x3 m; // tangent to world matrix

m[0] = IN.TexCoord1;

m[1] = IN.TexCoord2;

m[2] = IN.TexCoord3;

float3 Nw = mul(m, N.xyz);

// float3 E = float3(IN.TexCoord1.w, IN.TexCoord2.w, IN.TexCoord3.w);

float3 E = IN.eyeVector;

float3 R = reflect(-E, Nw);

return texCUBE(EnvironmentMap, R);

}

float4 OceanPS20(v2f IN,

uniform sampler2D NormalMap,

uniform samplerCUBE EnvironmentMap,

uniform half4 deepColor,

uniform half4 shallowColor,

uniform half4 reflectionColor,

uniform half4 reflectionAmount,

uniform half4 waterAmount,

uniform half fresnelPower,

uniform half fresnelBias,

uniform half hdrMultiplier

) : COLOR

{

// sum normal maps

half4 t0 = tex2D(NormalMap, IN.bumpCoord0.xy)\*2.0-1.0;

half4 t1 = tex2D(NormalMap, IN.bumpCoord1.xy)\*2.0-1.0;

half4 t2 = tex2D(NormalMap, IN.bumpCoord2.xy)\*2.0-1.0;

half3 N = t0.xyz + t1.xyz + t2.xyz;

// half3 N = t1.xyz;

half3x3 m; // tangent to world matrix

m[0] = IN.TexCoord1;

m[1] = IN.TexCoord2;

m[2] = IN.TexCoord3;

half3 Nw = mul(m, N.xyz);

Nw = normalize(Nw);

// reflection

float3 E = normalize(IN.eyeVector);

half3 R = reflect(-E, Nw);

half4 reflection = texCUBE(EnvironmentMap, R);

// hdr effect (multiplier in alpha channel)

reflection.rgb \*= (1.0 + reflection.a\*hdrMultiplier);

// fresnel - could use 1D tex lookup for this

half facing = 1.0 - max(dot(E, Nw), 0);

half fresnel = fresnelBias + (1.0-fresnelBias)\*pow(facing, fresnelPower);

half4 waterColor = lerp(deepColor, shallowColor, facing);

return waterColor\*waterAmount + reflection\*reflectionColor\*reflectionAmount\*fresnel;

// return waterColor;

// return fresnel;

// return reflection;

}

technique PS20 <

string Script = "Pass=p0;";

> {

pass p0 <

string Script = "Draw=geometry;";

> {

VertexShader = compile vs\_2\_0 BumpReflectWaveVS(wvpMatrix, worldMatrix, viewInverseMatrix,

bumpHeight, textureScale, bumpSpeed, time,

waveFreq, waveAmp);

Zenable = true;

ZWriteEnable = true;

CullMode = None;

// PixelShader = compile ps\_2\_0 BumpReflectPS20(normalMapSampler, envMapSampler);

PixelShader = compile ps\_2\_0 OceanPS20(normalMapSampler, envMapSampler,

deepColor, shallowColor, reflectionColor, reflectionAmount, waterAmount,

fresnelPower, fresnelBias, hdrMultiplier);

}

}