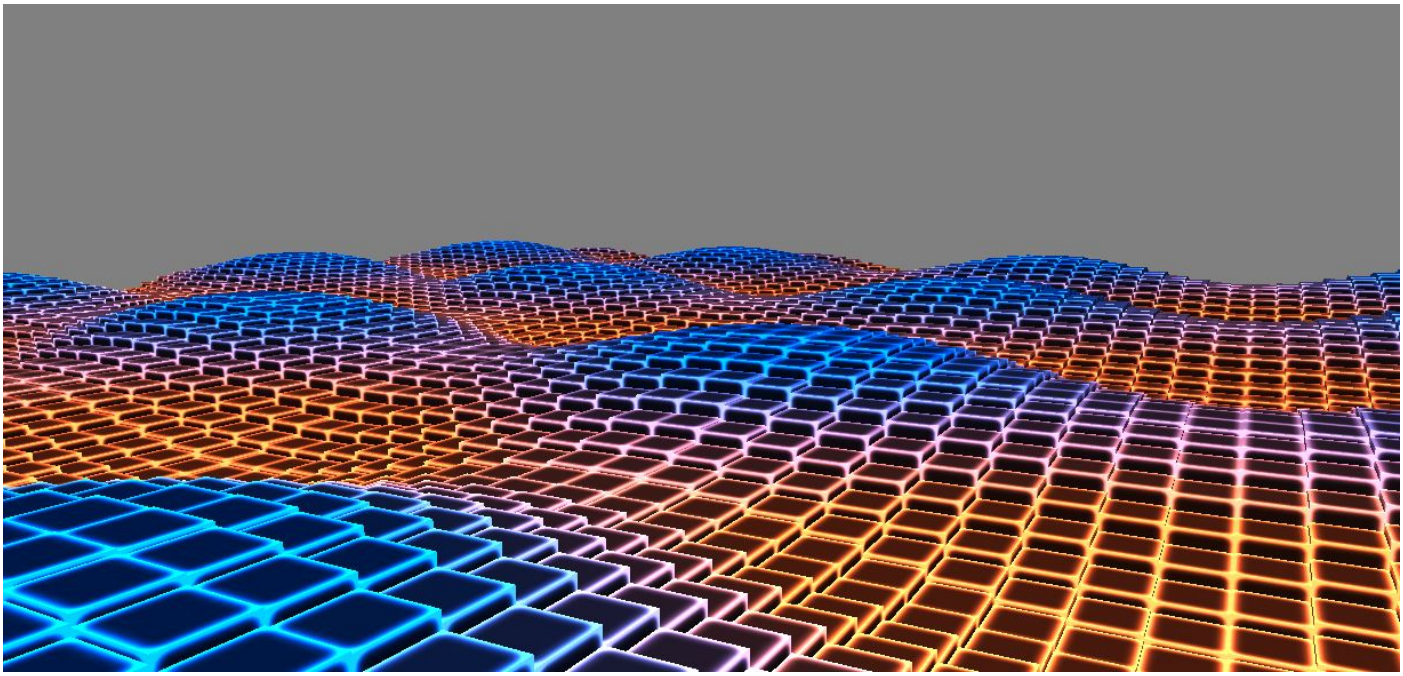
Geometry shader: Cyber Ocean



Introduction

The goal of this paper is to get familiar with the use of geometry shaders and add an interesting twist to a traditional ocean shader.

Using only a plane as a base mesh we will:

- Adjust the vertex positions to create a wave surface that resembles an ocean
- Adjust the color of the surface based on the height of the vertices
- Replace existing geometry with cubes at a certain height

Basic Waves

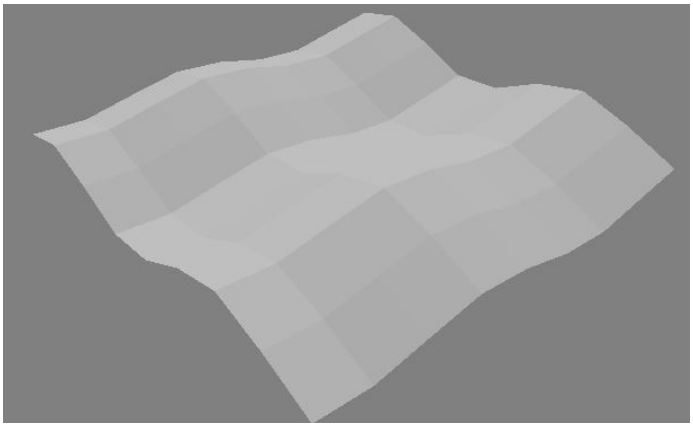


IMAGE 1 OCEAN PLANE

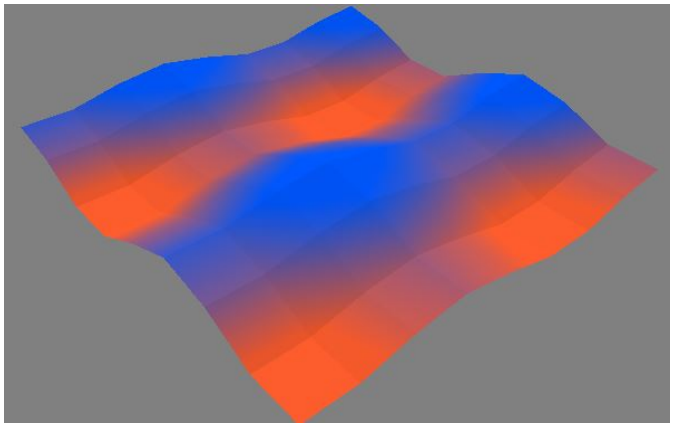


IMAGE 2 COLORIZED OCEAN PLANE

The first step into the creation of the ocean shader is to take our simple plane mesh and get it to behave like an ocean surface. In other words, to get the plane vertices to move up and down in a somewhat realistic wave like pattern.
To accomplish this, a sine function is used to offset the height value of each vertex that gets fed into the geometry shader. Note that this could also be done within the vertex shader. However since the purpose of this paper is to get familiar with geometry shaders we will recreate the original geometry in the geometry shader and apply the sine function to it.

The following line of pseudo code illustrates how the height value of the vertices will be altered.

*Vertex.Position.y +=Amplitude * sin(Speed*TIME + PeriodScale* Vertex.Position.xy);*

The Amplitude will determine the height of the waves, Speed controls the speed at which the waves will roll over the plane and the PeriodScale variable gives control over how many times the sine wave will be applied over a certain distance. The horizontal position of the vertex will ensure that the surface will follows the sine function instead of moving the entire plane up and down.
This function is applied to all the vertices twice. The first time using the x-position of the vertex and the second time using the z-position. This way , using different values for the parameters, an interesting wave pattern can be achieved.

```
[maxvertexcount(50)]
void WaveGenerator(triangle VS_DATA vertices[3], inout TriangleStream<GS_DATA> triStream)
{
    //Create Existing Geometry with a new height value
    vertices[0].Position.y +=xAmplitude * sin(xSpeed*m_Time +xPeriodScale*vertices[0].Position.x);
    vertices[1].Position.y +=xAmplitude * sin(xSpeed*m_Time+xPeriodScale*vertices[1].Position.x);
    vertices[2].Position.y +=xAmplitude * sin(xSpeed*m_Time+xPeriodScale*vertices[2].Position.x);

    vertices[0].Position.y +=zAmplitude * sin(zSpeed*m_Time+zPeriodScale*vertices[0].Position.z);
    vertices[1].Position.y +=zAmplitude * sin(zSpeed*m_Time+zPeriodScale*vertices[1].Position.z);
    vertices[2].Position.y +=zAmplitude * sin(zSpeed*m_Time+zPeriodScale*vertices[2].Position.z);

    float3 normal = cross(normalize(vertices[0].Position - vertices[1].Position),normalize(vertices[0].Position - vertices[2].Position));

    CreateVertex(triStream,vertices[0].Position,normal,vertices[0].TexCoord,vertices[0].Position.y);
    CreateVertex(triStream,vertices[1].Position,normal,vertices[1].TexCoord,vertices[1].Position.y);
    CreateVertex(triStream,vertices[2].Position,normal,vertices[2].TexCoord,vertices[2].Position.y);
}
```

IMAGE 3 BASIC WAVE GEOMETRY SHADER

```
//******************
// GEOMETRY SHADER *
//******************
void CreateVertex(inout TriangleStream<GS_DATA> triStream, float3 pos, float3 normal, float2 texCoord,float blockHeight)
{
    //Step 1. Create a GS_DATA object
    GS_DATA temp = (GS_DATA)0;

    //Step 2. Transform the position using the WVP Matrix
    //and assign it to (GS_DATA object).Position (Keep in mind: float3 -> float4)
    temp.Position = mul ( float4 (pos, 1.0f ), m_MatrixWorldViewProj );

    //Step 3. Transform the normal using the World Matrix
    //and assign it to (GS_DATA object).Normal (Only Rotation, No translation!)
    temp.Normal = mul(normal, (float3x3)m_MatrixWorld);

    //Step 4. Assign texCoord to (GS_DATA object).TexCoord
    temp.TexCoord = texCoord;

    //Step 5. Assign the height to (GS_DATA object).Height
    temp.Height =  blockHeight;

    //Step 6. Append (GS_DATA object) to the TriangleStream parameter (TriangleStream::Append(...))
    triStream.Append(temp);
}
```

IMAGE 4 CREATE VERTEX METHOD

Using the hlsl code illustrated in Image 3 and Image 4 each triangle of the original mesh is reconstructed, transformed and a new normal is calculated. The height value is also added to the vertex data.
This value can later be used in the pixel shader. The result of this can be seen in Image 2.
Other uses for the height value could be :

- overlay additional textures to add height based foam
- alter the translucency
- …

Creating new geometry

Having figured out how to achieve the wave effect on our plane we can go ahead and start creating new geometry in the shader.
The usual next step would be to take the supplied base mesh and use the geometry shader to triangulate this mesh. That way a smooth, fluent and detailed ocean surface can be achieved. However that is not the look that we are trying to accomplish. Instead we will replace the original geometry with entirely new geometry based on the information supplied by the original triangles. For every triangle the rectangular footprint will be calculated and subdivided into a certain amount of smaller rectangles. These smaller rectangles will then be used to create cube shapes that will act as our ocean surface.

Calculate the rectangular footprint

```
float xLength = vertices[1].Position.x - vertices[0].Position.x;
//Handle different winding of tris->ensure that we have a workable width value
if(xLength==0)
{
    xLength = vertices[2].Position.x - vertices[0].Position.x;
}


float zLength = vertices[1].Position.z - vertices[0].Position.z;
//Handle different winding of tris->ensure that we have a workable depth value
if(zLength==0)
{
    zLength=vertices[2].Position.z - vertices[0].Position.z;
}
```

IMAGE 5 CALCULATE HORIZONTAL FOOTPRINT OF A TRIANGLE

Using the position of the vertices, the width and depth are calculated (Image 5). As the winding of the supplied mesh can't be known in advance, we need to capture the situations where the calculated length would be 0. The downside of this is that no check is being performed to prevent duplicated geometry creation. This would be a possible optimization and/or extension of this shader.

Creating the cubes

Using a double for loop the footprint of each triangle gets subdivided into a certain amount of new rectangles and a top quad is generated.

```
float3 newQuad[4];
float3 newQuad2[4];
float width = xLength/xDevisions;
float depth = zLength/zDevisions;

newQuad[0] = vertices[0].Position + float3(i*width,0,j*depth);
newQuad[1] = vertices[0].Position + float3((i+1)*width,0,j*depth);
newQuad[2] = vertices[0].Position + float3(i*width,0,(j+1)*depth);
newQuad[3] = vertices[0].Position + float3((i+1)*width,0,(j+1)*depth);
```

IMAGE 6 CALCULATE TOPQUAD VERTICES

Next, the center of this new quad is determined and used as the position that will be used to calculate the height of the cube. In addition to the sine formula illustrated in Image 3, some additional calculations are added to add some extra "randomness" to our surface.

```
//calculate the center of the new quad
float3 center = (newQuad[0] + newQuad[1] + newQuad[2] + newQuad[3])/4;

//calculate and add the xwave height
float heightValue =0;
heightValue += xAmplitude* sin(xSpeed*m_Time +xPeriodScale*center.x);

//add some variation to the heightvalue
heightValue += xAmplitude* sin(zSpeed*0.5*m_Time +xPeriodScale*center.y*5);
heightValue *=cos(xSpeed*m_Time*0.001);

//calculate and add the z wave height
heightValue +=zAmplitude * sin(zSpeed*m_Time +zPeriodScale*center.z);

//add some variation to the heightvalue
//heightValue += zAmplitude* sin(xSpeed*0.5*m_Time +zPeriodScale*center.y*5);
//heightValue *=cos(zSpeed*m_Time*0.001);

//apply the height to the new surface
newQuad[0].y +=heightValue;
newQuad[1].y +=heightValue;
newQuad[2].y +=heightValue;
newQuad[3].y +=heightValue;
```

IMAGE 7 CALCULATE CUBE HEIGHT

The only thing that remains now is to calculate the vertices that create the bottom plane and then use all 8 vertices to construct each face of the cube. The bottom plane vertices can easily be obtained by subtracting the box height value from the top vertices. To ensure a nice and clean shader effect, the box height is determined by taking the largest amplitude that is being used. This prevents the possibility of holes in our surface when a larger amplitude is being used. It is also important to determine a correct new normal direction for each surface that gets calculated, as well as the texture coordinates of each vertex. Image 8 illustrates how the back face is created and how a new face normal can be calculated.(newQuad = top vertices, newQuad2 = bottom vertices)

```
//back
normal = cross(normalize(newQuad2[0]-newQuad[1]),normalize(newQuad[0] - newQuad[1]));
normal = normalize(normal);
triStream.RestartStrip();
CreateVertex(triStream,newQuad[0],normal, float2(0,0),heightValue);
CreateVertex(triStream,newQuad[1],normal, float2(1,0),heightValue);
CreateVertex(triStream,newQuad2[0],normal,float2(0,1),heightValue);
CreateVertex(triStream,newQuad2[1],normal,float2(1,1),heightValue);
```

IMAGE 8 CREATE THE BACK CUBE FACE

The result of all of this can be seen in Image 9 and Image 10.

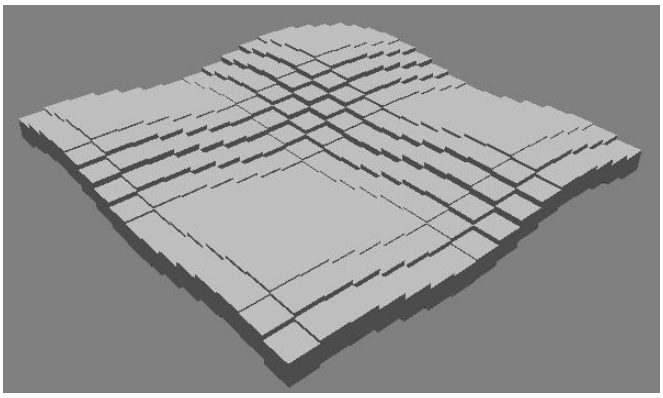


IMAGE 9 CUBE OCEAN

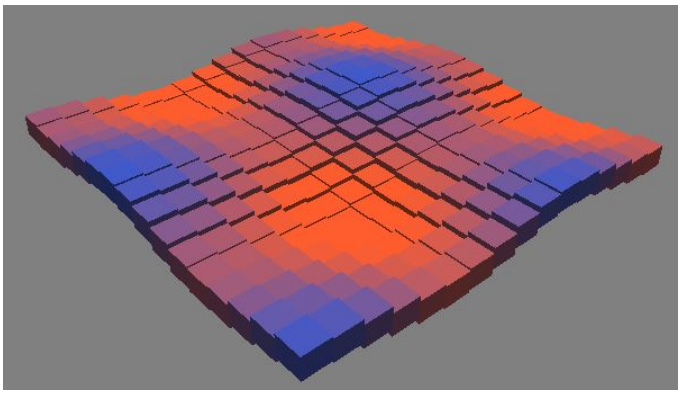


IMAGE 10 COLORIZED CUBE OCEAN

Pixel Shader

The final step into creating the look we are trying to achieve lies in the pixel shader.
A texture gets applied to the faces of the newly generated geometry, some simple shading is calculated and the material gets colorized based on the height of the cube. To finish it off, emissive is added to highlight the edges of each cube and achieve the final CyberOcean effect.

```
//***************
// PIXEL SHADER *
//***************
float4 MainPS(GS_DATA input) : SV_TARGET
{
    input.Normal=-normalize(input.Normal);

    //sample
    float alpha = m_TextureDiffuse.Sample(samLinear,input.TexCoord).a;
    float3 diffuse = m_TextureDiffuse.Sample( samLinear,input.TexCoord ).rgb;

    //shading
    float diffuseStrength = dot(input.Normal,m_LightDir);
    diffuseStrength = (diffuseStrength+1)/2;
    diffuseStrength = saturate(diffuseStrength);//clamp range to from 0 to 1
    diffuseStrength = pow(diffuseStrength,1);

    //Calculate Heightcolor weight
    float colorweight = (input.Height-m_ColorLowHeight)/(m_ColorHighHeight -m_ColorLowHeight);
    colorweight = saturate(colorweight);
    float3 HeightColor = lerp(m_ColorLow,m_ColorHigh,colorweight);

    //Colortint the diffuse with the height color
    float3 color = diffuse*diffuseStrength * HeightColor;

    //Contrast the diffuse and use this as emissive
    color += HeightColor* pow(diffuse,m_DiffuseToEmmisivePower)*m_EmissiveMultiplier;

    return float4(color,alpha);
}
```

IMAGE 11 PIXEL SHADER

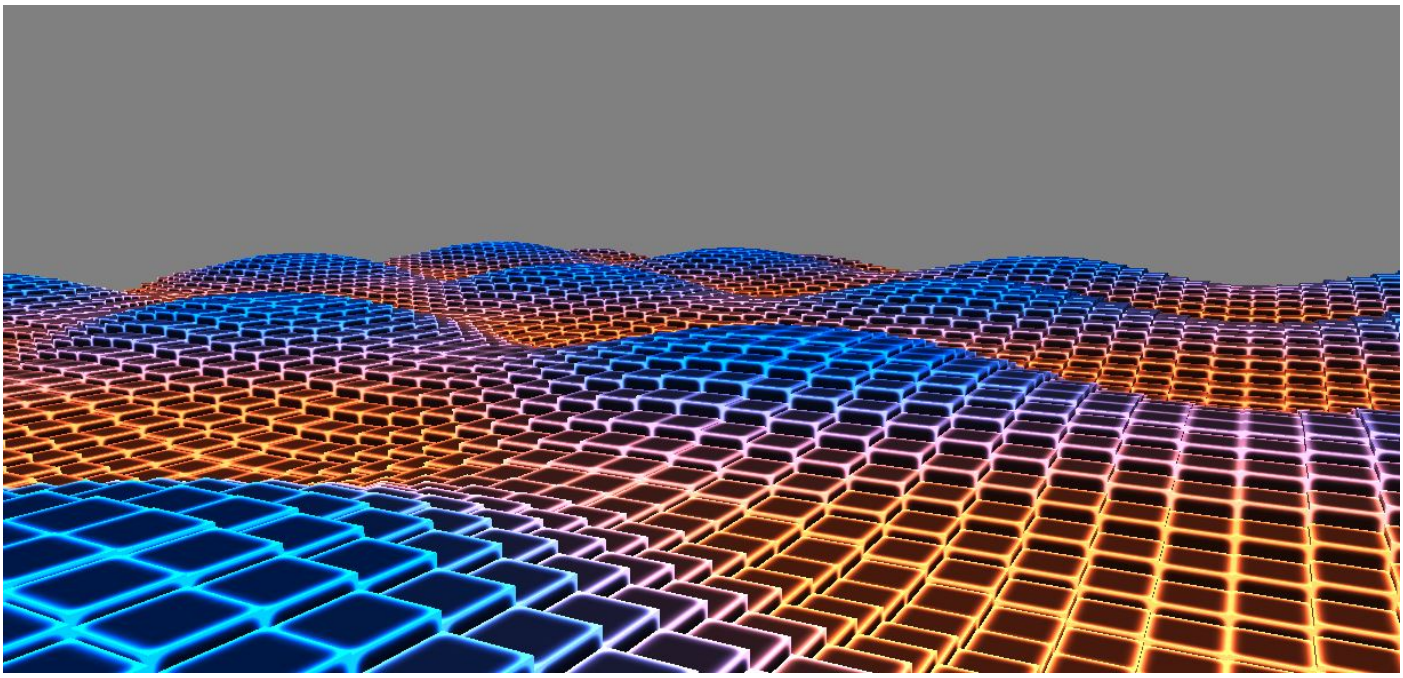


IMAGE 12 FINALIZED CYBER OCEAN 1

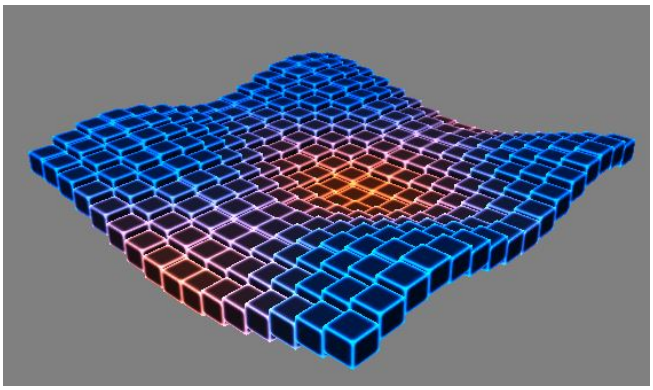


IMAGE 13 FINALIZED CYBER OCEAN 2

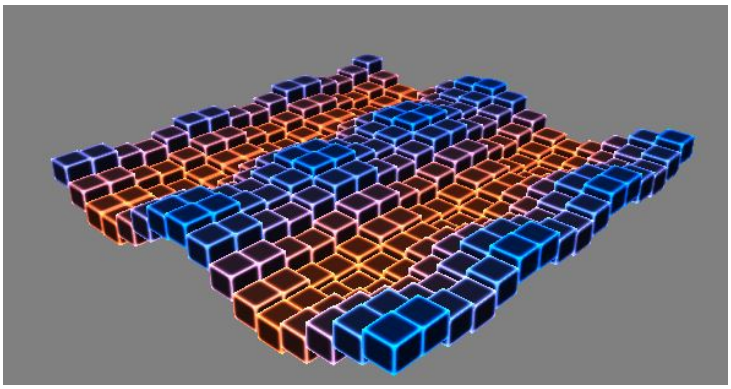


IMAGE 14 FINALIZED CYBER OCEAN 3