# Introduction

In this tutorial we will learn how to create a new material with custom rendering for using on any Wave Engine game.

# Getting started

Start by creating a new Wave Engine Game Project in Visual Studio. Add an entity that contains a test shape to the scene (we will be using a sphere); this way, we will be able to see how our new material behaves when we implement it:

Entity testShape = new Entity("TestShape")

.AddComponent(new Transform3D())

.AddComponent(Model.CreateSphere(5, 32))

.AddComponent(new MaterialsMap(new MyMaterial("Content/DefaultTexture.wpk")))

.AddComponent(new ModelRenderer());

We have purposely created the sphere with a bigger tessellation because in the sample shader we are going to perform a displacement transformation and without enough vertices it would look ugly.

# Creating the material

Add a new class to your Game Library project called MyMaterial and make it inherit from WaveEngine.Framework.Graphics.Material. You will notice that Visual Studio warns you that the class must implement two members, CurrentTechnique and Initialize. These members, along with SetParameters, are obligatory for each custom material and we will explain later what does each one.

## Defining shader techniques

We will start by creating a static array of ShaderTechnique objects. Inside each of them we will store the parameters that the adapter will need to create the internal shader objects: the technique name, the vertex and pixel shader file names and a VertexFormat with the same layout of the input structure of the vertex shader.

private static ShaderTechnique[] techniques =

{

new ShaderTechnique("MyMaterialTechnique",

"MyMaterialvs",

"MyMaterialps",

VertexPositionNormalTexture.VertexFormat),

};

This way, when we need to initialize each technique, we will have all of its properties already stored for easily accessing them.

## Declaring shader parameters

We need to create a structure that will hold all the parameters accessed by the shader. Since it is directly mapped to a buffer, we need to specify its StructLayout as LayoutKind.Sequential; and due to technical limitations of DirectX, it must have a Size multiple of 16 bytes (although the total size of the contained members can be less than that). Then, declare a private member variable that will hold an instance of the struct containing the parameters.

[StructLayout(LayoutKind.Sequential, Size = 16)]

private struct MyMaterialParameters

{

public float Time;

}

private MyMaterialParameters shaderParameters = new MyMaterialParameters();

## Adding a texture map

Since this sample material shows how texture mapping is done, we are going to need a property that stores a handle to an existing Texture object. Then, there are two ways to construct the material with the specified texture:

* Pass a Texture object as a parameter to the constructor.
* Pass a string that contains the path of the Texture asset and the material will load it when needed.

We are going to illustrate the second method, so add a string field and a Texture property:

private string diffuseMapPath;

public Texture DiffuseMap

{

get;

set;

}

We will show later how to initialize them.

## Selecting the appropriate technique

Remember that CurrentTechnique field that was mentioned previously? It is a read-only field that returns the name of the shader’s technique that should be used when drawing depending on how the material is configured (is lighting enabled? Should I draw using a texture?). Since this sample material only has one technique, this will be pretty straightforward:

public override string CurrentTechnique

{

get { return techniques[0].Name; }

}

## Laying out the constructor

The constructor of the custom material takes care of initializing the default values of the material. The only requirements that you must always meet is assigning the private instance of the struct containing the parameters to the Parameters property. This is done so DirectX can properly map the structure’s layout to its internal buffers when creating the shader object. After this, you can safely call InitializeTechniques passing the array of ShaderTechnique previously defined as the only parameter:

public MyMaterial(string diffuseMap)

: base(DefaultLayers.Opaque)

{

this.diffuseMapPath = diffuseMap;

this.Parameters = this.shaderParameters;

this.InitializeTechniques(techniques);

}

## Initializing specific assets

The function Initialize takes care of initializing any members that couldn’t be done in the constructor; for example, Texture assets need to be loaded into an AssetsContainer for properly managing their lifetime. We are going to load here the texture that is needed for our shader:

public override void Initialize(AssetsContainer assets)

{

try

{

this.DiffuseMap = assets.LoadAsset<Texture2D>(this.diffuseMapPath);

}

catch (Exception e)

{

throw new InvalidOperationException("MyMaterial needs a valid texture.");

}

}

## Passing parameters to the shader

The function SetParameters passes any necessary data to the shader. You must perform these actions in the specified order:

* Call base.SetParameters
* Change any shader parameters in the private struct instance previously created (in this case, shaderParameters).
* Assing the struct instance to the Parameters field.
* Set any textures you wish to use in the correct texture slots.

public override void SetParameters()

{

base.SetParameters();

this.shaderParameters.Time = (float)DateTime.Now.TimeOfDay.TotalSeconds;

this.Parameters = shaderParameters;

this.graphicsDevice.SetTexture(this.DiffuseMap, 0);

}

## Writing the DirectX shader

Now that we have all the code to use our material inside Wave, we must write the shaders that will be called. We will start with the DirectX one.

We start by adding a Shaders folder to the project that contains the material class. This is mandatory as directory names are hardcoded inside Wave’s material handling logic. Inside this directory, we create one named HLSL and other named GLSL. Both directories will contain as many folders as materials we are creating, with the same name as the material’s class.

As for the HLSL shader, create a new .fx file where the shader source will be written. This file is only used as an intermediate step because Wave needs the shader in binary form, so name it as you like – we will be using MyMaterial.fx.

Start by adding the following code:

cbuffer Matrices : register(b0)

{

float4x4 WorldViewProj : packoffset(c0);

float4x4 World : packoffset(c4);

float4x4 WorldInverseTranspose : packoffset(c8);

};

This buffer is mandatory to all shaders as it contains the matrices automatically mapped by Wave. If you need additional ones, they can be passed on the Parameters buffer.

cbuffer Parameters : register(b1)

{

float Time : packoffset(c0.x);

};

This is the buffer that maps the custom parameters passed to the shader. Remember to lay them in the appropriate order and use the packoffset directive as needed.

Texture2D DiffuseTexture : register(t0);

SamplerState DiffuseTextureSampler : register(s0);

The texture map and sampler are mapped to the same slot that you specified via code, in this case, the first one.

struct VS\_IN

{

float4 Position : POSITION;

float3 Normal : NORMAL0;

float2 TexCoord : TEXCOORD0;

};

struct VS\_OUT

{

float4 Position : SV\_POSITION;

float2 TexCoord : TEXCOORD0;

};

Check that the vertex shader input structure matches the vertex format you specified on the shader technique declaration – in this case, VertexPositionNormalTexture.

Now, we will proceed to write the vertex and pixel shader functions. The vertex shader will apply a simple sine deformation based on the Time parameter passed, and the pixel shader will sample from the texture and map it to the surface:

VS\_OUT vsMyMaterial( VS\_IN input )

{

VS\_OUT output = (VS\_OUT)0;

float offsetScale = abs(sin(Time + (input.TexCoord.y \* 16.0))) \* 0.25;

float4 vectorOffset = float4(input.Normal, 0) \* offsetScale;

output.Position = mul(input.Position + vectorOffset, WorldViewProj);

output.TexCoord = input.TexCoord;

return output;

}

float4 psMyMaterial( VS\_OUT input ) : SV\_Target0

{

return DiffuseTexture.Sample(DiffuseTextureSampler, input.TexCoord);

}

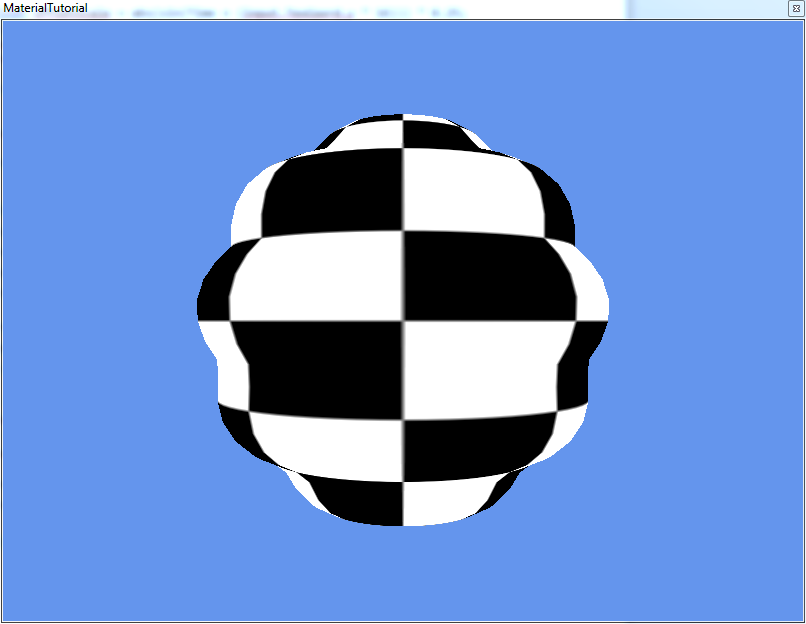
## Compiling the DirectX shader

Since HLSL shaders need to be compiled, we are going to use the fxc.exe tool for this. If you are on Windows 7, you will need to install the latest DirectX SDK, and you can find it on the Utilities\bin\x86 directory. If you are on Windows 8, you need to install the Windows 8 SDK and find the tool in the Windows Kits\8.0\bin\x86 directory.

Now, remember that you must compile the shaders using the shader model 4.0 – DirectX 9.1 level target profile so the same shader can be used across Windows, Windows Phone and Windows Store builds:

fxc.exe /nologo MyMaterial.fx /T vs\_4\_0\_level\_9\_1 /E vsMyMaterial /Fo MyMaterialvs.fxo  
fxc.exe /nologo MyMaterial.fx /T ps\_4\_0\_level\_9\_1 /E psMyMaterial /Fo MyMaterialps.fxo

Add the output files to the HLSL/MyMaterial directory of your project and remember to set the Build Action as Embedded Resource. Now try launching your project and you will see the result if there are no errors:



## Writing the OpenGL shader

Now, it’s time to add the shader for OpenGL platforms. Add the vertex (MyMaterialvs.vert) and fragment (MyMaterialps.frag) files to the GLSL/MyMaterial folder and write the code on them. Remember to put their Build Action to Embedded Resource.

uniform mat4 WorldViewProj;

uniform float Time;

attribute vec3 Position0;

attribute vec3 Normal0;

attribute vec2 TextureCoordinate0;

varying vec2 outTexCoord;

void main(void)

{

float offsetScale = abs(sin(Time + (TextureCoordinate0.y \* 16.0))) \* 0.25;

vec3 vectorOffset = Normal0 \* offsetScale;

gl\_Position = WorldViewProj \* vec4(Position0 + vectorOffset, 1);

outTexCoord = TextureCoordinate0;

}

Note that, since OpenGL doesn’t have support for variable buffers right now, the parameters of the shader are specified as uniform variables and the format of the vertex buffer as attributes. Apart from that, the shader is very similar to the one made in HLSL.

#ifdef GL\_ES

precision mediump float;

#endif

uniform sampler2D Texture;

varying vec2 outTexCoord;

void main(void)

{

gl\_FragColor = texture2D(Texture, outTexCoord);

}

Now, there is a catch with how textures work in the OpenGL version: make sure you name your uniform sampler2D with the same name as the texture’s property you set in the MyMaterial class. Otherwise, unexpected errors could happen.

Now, convert your project to Android and run it. In case there are any errors compiling the new shaders, they will be written on the Output window of Visual Studio.