White Paper

Texture Arrays Terrain Rendering

February 2007
WP-03015-001_v01

Document Change History

Version	Date	Responsible	Reason for Change
_v01	February 20, 2007	BD, TS	Initial release

Go to sdkfeedback@nvidia.com to provide feedback on Texture Arrays.

Texture Arrays

Abstract

The appearance of terrain varies widely. When drawing large expansive terrain, artists require many different textures. There are different ways to solve this, including splitting the mesh into many pieces with different bound textures per draw call. DirectX10 introduces the concept of texture arrays allowing an array of textures to be bound to a shader as a single piece of state, dynamically indexed within the shader. This allows a palette of textures to be applied to a terrain mesh, which means we are able to draw the entire terrain section in a single draw call.

The equivalent in DirectX9 would require multiple draw calls which would reduce efficiency. Alternatively, texture *atlases* are commonly used. However, atlases require careful handling and impose restrictions on art content, so are not ideal (see references).

Bryan Dudash
NVIDIA Corporation

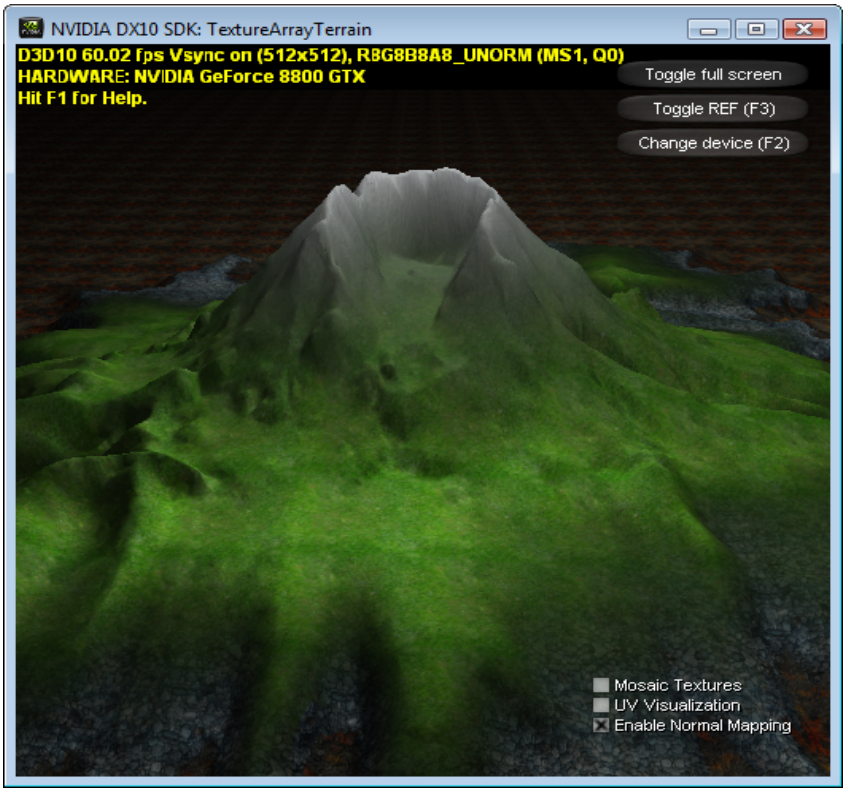

Figure 1. Terrain Mesh Rendered with Four Blended Textures all Read from a Texture Array

Motivation

The motivation for the texture arrays technique is performance. In general, reducing state changes and draw calls reduces the overhead on the CPU, which provides more processing power to the application. CPU overhead is a common bottleneck in modern 3D applications. This technique allows a developer to reduce overhead while maintaining robustness and unique textures for rendered polygons.

A secondary motivation, compared with texture atlases, is to simplify a game's art pipeline. Use of an atlas requires changes to the model's texture coordinates to reflect a texture's position within the atlas. This complicates the art pipeline. Also, packing textures into adjacent space in an atlas leads to blending artifacts if handled naively, and makes it difficult to support tiled textures. These issues all go away when using texture arrays.

How It Works

The way the sample makes use of texture arrays is quite simple. Each vertex of the terrain mesh includes three texture coordinate vectors. Each of these coordinates is a 4-component vector. This is in contrast to the standard 2-component UV. The third component contains the array index into the texture array and the fourth component contains an alpha blend value. Then at runtime, in the shader, we index into the texture array to pull out the appropriate texture and use the alpha blend value to blend together multiple layers.

Note: The sample does not provide a way to export or author multiple texture coordinates into your vertex data. It is assumed that developers will have an established tools pipeline and are able to implement that functionality themselves.

The three sets of coordinates give us support for blending of two *base* textures per vertex as well as a *decal* texture to be overlaid on top of the blend result. The equation for this is:

```
blendedColor = color1 * color1alpha + color2 * color2alpha;
if(decalAlpha > 0)
finalColor = decalColor * decalAlpha + blendedColor * (1-decalAlpha);
else
      finalColor = blendedColor;
```

The shader code calculating a **`finalColor`** is:

```
// Load the color values (3rd component is the slice of the aray from
which to sample)
// We multiply by the last component of the texcoord, which is set by
the application to be the blend weight
float4 albedo0 =
g_txTextures.Sample(g_samAniso,input.tex0.xyz)*input.tex0.w;
float4 albedo1 =
g_txTextures.Sample(g_samAniso,input.tex1.xyz)*input.tex1.w;

float4 bumps0 =
g_txNormals.Sample(g_samAniso,input.tex0.xyz)*input.tex0.w;
float4 bumps1 =
g_txNormals.Sample(g_samAniso,input.tex1.xyz)*input.tex1.w;

// combine using sampled alphas
float3 finalAlbedo = (albedo0.xyz * albedo0.w) + (albedo1.xyz *
albedo1.w);
float3 finalBumps = bumps0.xyz + bumps1.xyz;

// third texture is the decal, but is optional
if(input.tex2.w > 0)
{
      float4 decal = g_txTextures.Sample(g_samAniso,input.tex2.xyz);
      float4 decalBumps =
g_txTextures.Sample(g_samAniso,input.tex2.xyz);
      float decalAlpha = input.tex2.w * decal.w;
      finalAlbedo = finalAlbedo*(1-decalAlpha) + decal.xyz*decalAlpha;
```

```
      finalBumps = normalize(finalBumps*(1-decalAlpha) +
decalBumps*decalAlpha);
}
```

We use a conditional since, in the general case, the decal texture will not be used. Also, the decal might not even be needed depending on the application. Also, notice that we multiply each diffuse color by its alpha value. This allows the artists to have per-texel control over the coverage of a terrain texture. If the artist wants to control the blending using the alpha value of the texture, they can.

Texture Arrays

The DirectX10 API for creating a texture array is similar to the one for creating a regular texture. The difference is in the **`D3D10_TEXTURE2D_DESC`** structure where the **`ArraySize`** element is set to the size of the array. Refer to the code block below for an example of creating a texture array.

```
D3D10_TEXTURE2D_DESC desc;
desc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
desc.Width = iWidth;
desc.Height = iHeight;
desc.MipLevels = iMIPLevels;
desc.Usage = D3D10_USAGE_DEFAULT;
desc.BindFlags = D3D10_BIND_SHADER_RESOURCE;
desc.CPUAccessFlags = 0;
desc.ArraySize = iNumTextures;
pd3dDevice->CreateTexture2D( &desc, NULL, ppTex2D);
```

In the sample code, the function **`MultiTextureTerrain::loadTextureArray()`** shows how to create a texture array and fill it with data from an array of DDS files.

Texture Ping Pong

An issue will arise with texture coordinates that is worth mentioning. If a mesh uses three base textures—let's say in a progressive blend from texture **`0`** thru **`2`** (see Figure 2) then in order to get proper interpolation of the texture coordinated between vertices you will need to swap the texture coordinate that the shader texture is using as the texture changes. The system doesn't understand which texture will be read with each **`texcoord`**,

For this example, your textures are red brick, green grass and white snow. Your mesh transitions from those in that order as shown in Figure 2. You need to be careful that all meshes in the contiguous area of green assign the green texture to be read from **`texcoord 1`**. This may seem obvious here, but if you adopt a strategy of always putting the texture transitioning from in **`texcoord 0`** and the texture transitioning to into **`texcoord 1`**, then you will get improper blending at the transition areas.

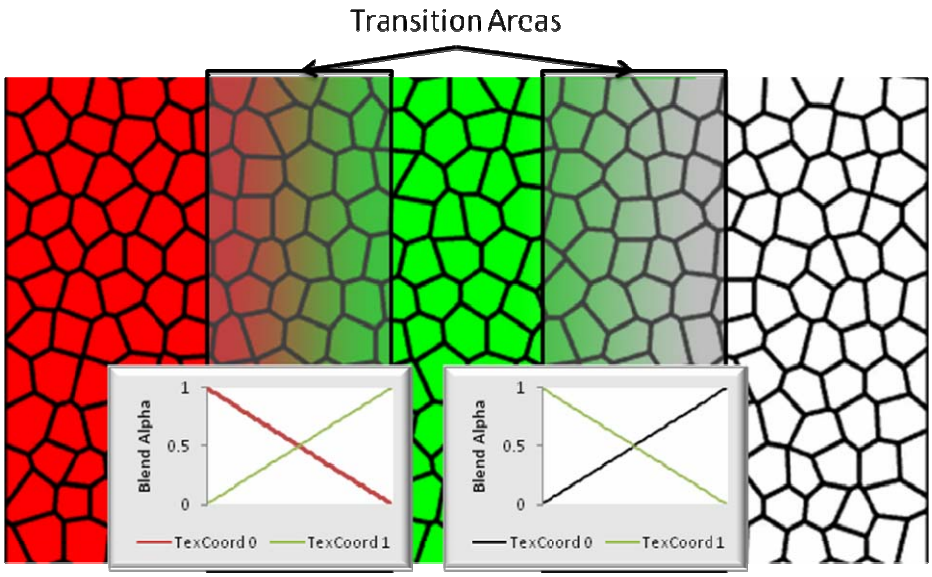

Figure 2. Example of Texture Ping Pong

Implementation Details

This project is extremely simple. The source code is divided into two **cpp** files and an fx file:

- **TextureArrayTerrain.cpp**: This is the standard sample DX10 setup code and UI code. It simply creates a **MultiTextureTerrain** object and calls into it.
- **MultiTextureTerrain.cpp**: This file contains all the code to create and render a terrain mesh. It loads the mesh file, and all relevant textures, and setups render state.
- **TextureArrayTerrain.fx** contains the D3DX effect.

Running the Sample

The arrow keys move the camera around in a FPS-style control scheme. The mouse rotates the camera view. **Esc** exits.

Performance

The GPU load of this technique will be on par with separate draw calls for the sample rendering. The performance difference here is in the fact that there are less state changes and only a single draw call to render the whole terrain mesh.

Also note that we draw the terrain in two passes. The first pass is a **`z`** only pass that should run at blazing speeds. This engages the **`z`** cull hardware of modern graphics processor allowing the hardware to efficiently cull out occluded surfaces.

Integration

Using this technique in your game engine requires that your export tool path include multiple texture coords for each terrain vertex, as well as moving to a **`float4 texcoord`** system. The tool should also support inserting the relevant index into a texture array to allow selection of the proper texture.

In addition, similar to the process the sample uses, your codebase will need to load all textures for the terrain palette and insert them into a texture array object. This object can then be used instead of separate textures.

References

- *NVIDIA SDK9.5 Texture Atlases Sample Whitepaper*
- *Improving Batching using Texture Atlases,* Mattias Wloka

.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com