# 11.3 纹理数组

## 11.3.1 概述

纹理数组（texture array）对象用于存储一个纹理阵列。在C++代码中，纹理数组对象由**ID3D11Texture2D**接口表示（该接口也用于表示单个纹理对象）。其实，在创建**ID3D11Texture2D**对象时有一个称为**ArraySize**的属性可以设置所要存储的纹理对象的数量。不过，我们总是使用D3DX来创建纹理，所以不必直接设置这个数据成员。在effect文件中，纹理数组对象由**Texture2DArray**类型表示：

Texture2DArray gTreeMapArray;

现在，你可能不明白我们为什么要使用纹理数组对象。为什么不这样用：

Texture2D TexArray[4];

...

float4 PS(GeoOut pin) : SV\_Target

{

float4 c = TexArray[pin.PrimID%4].Sample(samLinear, pin.Tex);

这会引发一个错误：“采样器数组索引必须是一个常量表达式”。换句话说，它不希望数组索引随着像素而变化。当我们指定一个常量数组索引时，该语句可以正常运行：

float4 c = TexArray[2].Sample(samLinear, pin.Tex);

但是与第一种方案相比，这条语句没多大用处。

## 11.3.2 对纹理数组进行采样

在树广告牌演示程序中，我们用如下代码对纹理数组进行采样：

float3 uvw = float3(pin.Tex, pin.PrimID%4);

float4 diffuse = gTreeMapArray.Sample(samLinear,uvw);

当使用纹理数组时，我们需要3个纹理坐标。前两个纹理坐标是普通的2D纹理坐标；第3个纹理坐标是纹理数组的索引。例如，0.0是数组中的第1个纹理的索引，1.0是数组中的第2个纹理的索引，2.0是数组中的第3个纹理的索引，以此类推。

在树广告牌演示程序中，我们使用了一个包含4个元素的纹理数组，每个元素带有一种不同的树图像 （参见图11.7）。不过，我们绘制的图元数量不只4个，图元ID的值会大于 3。所以，我们让图元ID以4为模（pin.primID%4），把图元ID映射为0、1、2、3，得到一个有效的数组索引。

****

**图11.7 树广告牌图像。**

使用纹理数组的好处之一是，我们可以在一次绘图调用中使用不同的纹理来绘制一组图元。以前，我们必须这样做（伪代码）：

SetTextureA();

DrawPrimitivesWithTextureA();

SetTextureB();

DrawPrimitivesWithTextureB();

...

SetTextureZ();

DrawPrimitivesWithTextureZ();

每个Set和Draw调用都会带来一些额外的系统开销。通过使用纹理数组，我们可以将其简化为一次Set调用和一次Draw调用：

SetTextureArray();

DrawPrimitivesWithTextureArray();

## 11.3.3 载入纹理数组

在撰写本书时，Direct3D并没有提供将一批图像文件同时载入纹理数组的D3DX函数。所以，我们必须自己完成一任务。实现过程归纳如下：

1．分别创建每个纹理对象。

2．创建纹理数组对象。

3．将每个纹理对象依次复制到纹理数组元素中去。

4．为纹理数组对象创建一个着色器资源视图。

我们在*d3dUtil.h/cpp*中实现了一个辅助函数用来从一个文件名列表创建纹理数组。这些纹理的大小要求是相同的。具体的实现代码如下。

ID3D11ShaderResourceView\* d3dHelper::CreateTexture2DArraySRV(

ID3D11Device\* device, ID3D11DeviceContext\* context,

std::vector<std::wstring>& filenames,

DXGI\_FORMAT format,

UINT filter,

UINT mipFilter)

{

//

// 从文件加载单独的纹理元素。这些纹理不能被GPU使用(0 bind flags)，

// 只是用来从文件中加载图像数据。我们使用STAGING让CPU可以访问资源。

//

UINT size = filenames.size();

std::vector<ID3D11Texture2D\*> srcTex(size);

for(UINT i = 0; i < size; ++i)

{

D3DX11\_IMAGE\_LOAD\_INFO loadInfo;

loadInfo.Width = D3DX11\_FROM\_FILE;

loadInfo.Height = D3DX11\_FROM\_FILE;

loadInfo.Depth = D3DX11\_FROM\_FILE;

loadInfo.FirstMipLevel = 0;

loadInfo.MipLevels = D3DX11\_FROM\_FILE;

loadInfo.Usage = D3D11\_USAGE\_STAGING;

loadInfo.BindFlags = 0;

loadInfo.CpuAccessFlags = D3D11\_CPU\_ACCESS\_WRITE | D3D11\_CPU\_ACCESS\_READ;

loadInfo.MiscFlags = 0;

loadInfo.Format = format;

loadInfo.Filter = filter;

loadInfo.MipFilter = mipFilter;

loadInfo.pSrcInfo = 0;

HR(D3DX11CreateTextureFromFile(device, filenames[i].c\_str(),

&loadInfo, 0, (ID3D11Resource\*\*)&srcTex[i], 0));

}

//

// 创建纹理数组。每个纹理元素都具有相同的格式/大小。

//

D3D11\_TEXTURE2D\_DESC texElementDesc;

srcTex[0]->GetDesc(&texElementDesc);

D3D11\_TEXTURE2D\_DESC texArrayDesc;

texArrayDesc.Width = texElementDesc.Width;

texArrayDesc.Height = texElementDesc.Height;

texArrayDesc.MipLevels = texElementDesc.MipLevels;

texArrayDesc.ArraySize = size;

texArrayDesc.Format = texElementDesc.Format;

texArrayDesc.SampleDesc.Count = 1;

texArrayDesc.SampleDesc.Quality = 0;

texArrayDesc.Usage = D3D11\_USAGE\_DEFAULT;

texArrayDesc.BindFlags = D3D11\_BIND\_SHADER\_RESOURCE;

texArrayDesc.CPUAccessFlags = 0;

texArrayDesc.MiscFlags = 0;

ID3D11Texture2D\* texArray = 0;

HR(device->CreateTexture2D( &texArrayDesc, 0, &texArray));

//

// 将单独的纹理元素复制到纹理数组中。

//

// 处理每个纹理元素...

for(UINT texElement = 0; texElement < size; ++texElement)

{

// 处理每个渐进纹理层...

for(UINT mipLevel = 0; mipLevel < texElementDesc.MipLevels; ++mipLevel)

{

D3D11\_MAPPED\_SUBRESOURCE mappedTex2D;

HR(context->Map(srcTex[texElement], mipLevel, D3D11\_MAP\_READ, 0, &mappedTex2D));

context->UpdateSubresource(texArray,

D3D11CalcSubresource(mipLevel, texElement, texElementDesc.MipLevels),

0, mappedTex2D.pData, mappedTex2D.RowPitch, mappedTex2D.DepthPitch);

context->Unmap(srcTex[texElement], mipLevel);

}

}

//

// 创建纹理数组的资源视图

//

D3D11\_SHADER\_RESOURCE\_VIEW\_DESC viewDesc;

viewDesc.Format = texArrayDesc.Format;

viewDesc.ViewDimension = D3D11\_SRV\_DIMENSION\_TEXTURE2DARRAY;

viewDesc.Texture2DArray.MostDetailedMip = 0;

viewDesc.Texture2DArray.MipLevels = texArrayDesc.MipLevels;

viewDesc.Texture2DArray.FirstArraySlice = 0;

viewDesc.Texture2DArray.ArraySize = size;

ID3D11ShaderResourceView\* texArraySRV = 0;

HR(device->CreateShaderResourceView(texArray, &viewDesc, &texArraySRV));

//

// 清除--我们要的只是资源视图

//

ReleaseCOM(texArray);

for(UINT i = 0; i < size; ++i)

ReleaseCOM(srcTex[i]);

return texArraySRV;

}

**ID3D11DeviceContext::UpdateSubresource**方法通过CPU将一个子资源复制给另一个子资源（该方法的参数描述请参见SDK文档）。

void ID3D11DeviceContext::UpdateSubresource(

ID3D11Resource \*pDstResource,

UINT DstSubresource,

const D3D11\_BOX \*pDstBox,

const void \*pSrcData,

UINT SrcRowPitch,

UINT SrcDepthPitch);

**1．pDstResource**：目标资源对象。

**2．DstSubresource**：我们要更新的目标资源中的子资源索引（见11.3.4节）。

**3．pDstBox**：指向一个**D3D11\_BOX**实例的指针，指定要更新的目标子资源的大小，若设定为null则更新整个子资源。

**4．pSrcData**：指向源数据的指针。

**5．SrcRowPitch**：源数据一行的大小，以字节为单位。

**6．SrcDepthPitch**：源数据一个深度的大小，以字节为单位。

注意，对于2D纹理而言，**SrcDepthPitch**参数看起来是不必要的；但是这个方法还要用于更新3D纹理，这种情况下此参数可被视为一个2D纹理堆。

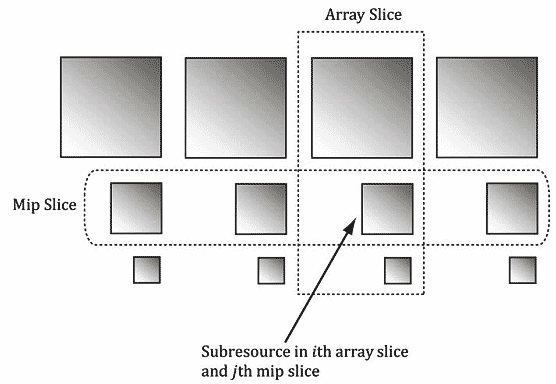
我们在初始化时调用前面的函数创建树图像的纹理数组：

ID3D11ShaderResourceView\* mTreeTextureMapArraySRV;

mTreeTextureMapArraySRV = d3dHelper::CreateTexture2DArraySRV(md3dDevice,md3dImmediateContext, treeFilenames ,DXGI\_FORMAT\_R8G8B8A8\_UNORM);

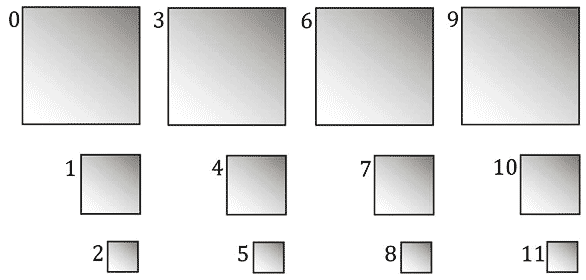
## 11.3.4 纹理子资源

现在我们已经讨论了纹理数组，下面我们来讨论子资源（subresources）的概念。图11.8展示了一个包含多个纹理的纹理数组。其中的每个纹理都有它自己的多级渐近纹理链。Direct3D API使用术语“数组切片（array slice）”表示一个完整的多级渐近纹理链中的一个元素，使用术语 “多级渐近切片（mip slice）”表示纹理数组中的所有多级渐近纹理链的特定层。“子资源”表示纹理数组元素中的单个多级渐近纹理层。

****

**图11.8 包含4个纹理的纹理数组。每个纹理带有3个多级渐近纹理层。**

只要给定一个数组切片索引和一个多级渐近切片索引，我们就可以访问纹理数组中的任何一个子资源。不过，子资源也可以通过线性索引来标识；Direct3D按照如图11.9所示的顺序标识线性索引。

****

**图11.9 通过线性索引标识纹理数组中的子资源。**

下面的工具函数用于计算线性子资源索引，它的3个参数分别表示多级渐近切片的索引、数组切片的索引和多级渐近纹理层的数量：

inline UINT D3D11CalcSubresource(

UINT MipSlice, UINT ArraySlice, UINT MipLevels);

使用的公式很简单：*k* = *ArraySlice*\**MipLevels* + *MipSlice*。