Week7

Geometry Shader

Hooman Salamat

Objectives

- Learn how to program geometry shaders.

- Discover how billboards can be implemented efficiently using the shader

- Recognize auto generated primitive IDs and some of their applications

- Find out how to create and use texture arrays.

- Understand how alpha-to-coverage helps with the aliasing problem of alpha cutouts

The Geometry Shader

The geometry shader stage is an optional stage that sits between the vertex and pixel shader stages.

While the vertex shader inputs vertices, the geometry shader inputs entire primitives.

For example, if we were drawing triangle lists, then conceptually the geometry shader program would be executed for each triangle T in the list:

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

OutputPrimitiveList = GeometryShader(T[i].vertexList);

A) the three vertices of each triangle are input into the geometry shader

B) The geometry shader outputs a list of primitives. The primitives output from the geometry shader are defined by a vertex list.

C) After the geometry shader stage, we have a list of vertices defining primitives in homogeneous clip space. These vertices are projected (homogeneous divide), and then rasterization occurs as usual.

D) Unlike Vertex Shader, Geometry Shader can create/destroy geometry. Vertex shader cannot create/destroy vertex.

❑ The input primitive can be expanded into one or more other primitives: a common application of the geometry shader is to expand a point into a quad

❑ The geometry shader can choose not to output a primitive based on some condition

PROGRAMMING GEOMETRY SHADERS

We must first specify the maximum number of vertices the geometry shader will output for a single invocation (the geometry shader is invoked per primitive).

The geometry shader takes two parameters: an input parameter and an output parameter.

The input parameter is always an array of vertices that define the primitive—one vertex for a point, two for a line, three for a triangle, four for a line with adjacency, and six for a triangle with adjacency.

The input parameter must be prefixed by a primitive type:

point: The input primitives are points.

line: The input primitives are lines (lists or strips).

triangle: The input primitives triangles (lists or strips).

lineadj: The input primitives are lines with adjacency (lists or strips).

triangleadj: The input primitives are triangles with adjacency (lists or strips).

```
[maxvertexcount(N)]

void ShaderName(

PrimitiveType InputVertexType InputName[NumElements],

inout StreamOutputObject<OutputVertexType>

OutputName)

{

// Geometry shader body…

}
```

PROGRAMMING GEOMETRY SHADERS

The output parameter always has the inout modifier.

The output parameter is always a stream type.

A stream type stores a list of vertices which defines the geometry the geometry shader is outputting.

A stream type is a template type, where the template argument is used to specify the vertex type of the outgoing vertices (e.g., GeoOut). There are three possible stream types:

PointStream<OutputVertexType>: A list of vertices defining a point list.

LineStream<OutputVertexType>: A list of vertices defining a line strip.

TriangleStream<OutputVertexType>: A list of vertices defining a triangle strip.

The input primitive into a geometry shader is always a complete primitive (e.g., two vertices for a line, and three vertices for a triangle). Therefore the geometry shader does not need to distinguish between lists and strips.

```
struct GeoOut
{
    float4 PosH     : SV_POSITION;
    float3 PosW     : POSITION;
    float3 NormalW  : NORMAL;
    float2 TexC     : TEXCOORD;
    uint   PrimID   : SV_PrimitiveID
};
[maxvertexcount(4)]
void GS(point VertexOut gin[1],
        uint primID : SV_PrimitiveID,
        inout TriangleStream<GeoOut> triStream){
```

For lines and triangles, the output primitive is always a strip. Line and triangle lists, however, can be simulated by using the intrinsic RestartStrip method:

Void StreamOutputObject<OutputVertexType>::RestartStrip();
Ends the current primitive strip and starts a new strip.

For example, if you wanted to output triangle lists, then you would call RestartStrip every time after three vertices were appended to the output stream.

PROGRAMMING GEOMETRY SHADERS

A geometry shader adds a vertex to the outgoing stream list using the intrinsic Append method:

```
Void
StreamOutputObject<OutputVertexType>::Append(OutputVertexType v);

GeoOut gout;

[unroll]
for(int i = 0; i < 4; ++i)
{
gout.PosH      = mul(v[i], gViewProj);

gout.PosW      = v[i].xyz;

gout.NormalW   = look;

gout.TexC      = texC[i];

gout.PrimID    = primID;


triStream.Append(gout);

}
```

Loop unrolling, also known as **loop unwinding**, is a loop transformation technique that attempts to optimize a program's execution speed at the expense of its binary size.

As a result of this modification, the new program has to make only 20 iterations, instead of 100. Afterwards, only 20% of the jumps and conditional branches need to be taken, and represents, over many iterations, a potentially significant decrease in the loop administration overhead.

Normal loop	After loop unrolling

int x;
 	
for (x = 0; x < 100; x++)
{
delete(x);
}
	int x;
	
	for (x = 0; x < 100; x += 5)
	 {
	delete(x);
	delete(x + 1);
	delete(x + 2);
	delete(x + 3);
	delete(x + 4);
	}

GS Examples

EXAMPLE 1:

GS ouputs at most 4 vertices.

The input primitive is a line.

The output is a triangle strip.

```
[maxvertexcount(4)]

void GS(line VertexOut gin[2],

inout TriangleStream<GeoOut> triStream)

{

// Geometry shader body…

}
```

EXAMPLE 2:

GS outputs at most 32 vertices.

The input primitive is a triangle.

 The output is a triangle strip.

```
[maxvertexcount(32)]

void GS(triangle VertexOut gin[3],

inout TriangleStream<GeoOut> triStream)

{

// Geometry shader body…

}
```

GS Example

```
struct VertexOut
{
float3 PosL : POSITION;
float3 NormalL : NORMAL;
float2 Tex : TEXCOORD;
};


struct GeoOut
{
float4 PosH : SV_POSITION;
float3 PosW : POSITION;
float3 NormalW : NORMAL;
float2 Tex : TEXCOORD;
float FogLerp : FOG;
};
```

EXAMPLE 3:

GS outputs at most 4 vertices.

The input primitive is a point.

The output is a triangle strip.

```
[maxvertexcount(4)]

void GS(point VertexOut gin[1],

inout TriangleStream<GeoOut> triStream)

{

// Geometry shader body…

}
```

Subdividing a triangle

Subdividing a triangle into four equally sized triangles. Observe that the three new vertices are the midpoints along the edges of the original triangle.

We can draw the subdivision in two strips:

Strip 1: bottom three triangles

Strip 2: top triangle

The following geometry shader illustrates the Append and RestartStrip methods; it inputs a triangle, subdivides it and outputs the four subdivided triangles.
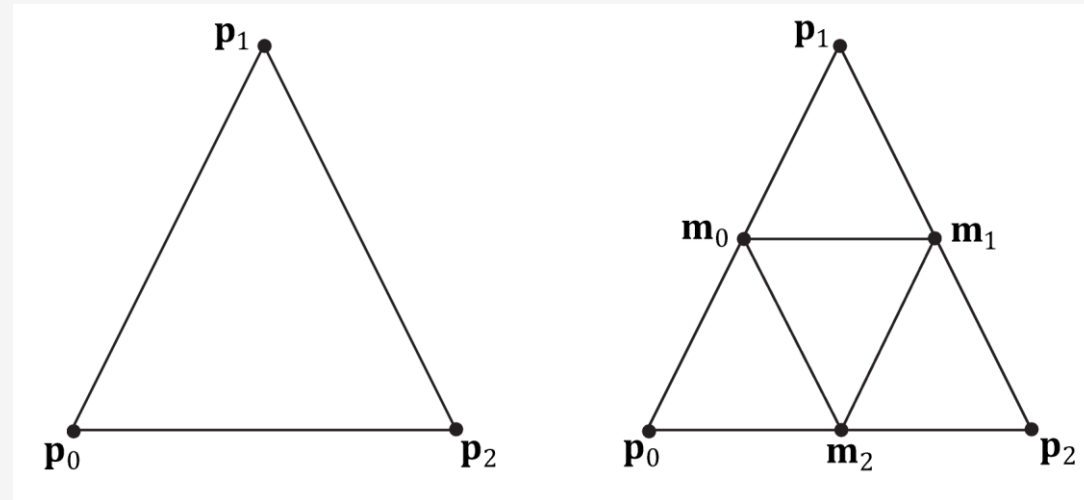
Code Sample

```
void Subdivide(VertexOut inVerts[3], out VertexOut outVerts[6])
{
VertexOut m[3];
// Compute edge midpoints.
m[0].PosL = 0.5f * (inVerts[0].PosL + inVerts[1].PosL);
m[1].PosL = 0.5f * (inVerts[1].PosL + inVerts[2].PosL);
m[2].PosL = 0.5f * (inVerts[2].PosL + inVerts[0].PosL);
// Project onto unit sphere
m[0].PosL = normalize(m[0].PosL);
m[1].PosL = normalize(m[1].PosL);
m[2].PosL = normalize(m[2].PosL);
// Derive normals.
m[0].NormalL = m[0].PosL;
m[1].NormalL = m[1].PosL;
m[2].NormalL = m[2].PosL;
// Interpolate texture coordinates.
m[0].Tex = 0.5f * (inVerts[0].Tex + inVerts[1].Tex);
m[1].Tex = 0.5f * (inVerts[1].Tex + inVerts[2].Tex);
m[2].Tex = 0.5f * (inVerts[2].Tex + inVerts[0].Tex);
outVerts[0] = inVerts[0];
outVerts[1] = m[0];
outVerts[2] = m[2];
outVerts[3] = m[1];
outVerts[4] = inVerts[2];
outVerts[5] = inVerts[1];
};
```

```
void OutputSubdivision(VertexOut v[6], inout TriangleStream<GeoOut> triStream)
{
GeoOut gout[6];
[unroll]
for (int i = 0; i < 6; ++i)
{
// Transform to world space space.
gout[i].PosW = mul(float4(v[i].PosL, 1.0f), gWorld).xyz;
gout[i].NormalW = mul(v[i].NormalL, (float3x3)gWorldInvTranspose);
// Transform to homogeneous clip space.
gout[i].PosH = mul(float4(v[i].PosL, 1.0f), gWorldViewProj);
gout[i].Tex = v[i].Tex;
}
[unroll]
for (int j = 0; j < 5; ++j) -> p0,m0,m2,m1,p2
{
triStream.Append(gout[j]); // Strip 1: bottom three triangles (5 vertices)
}
triStream.RestartStrip();  Strip 2: top triangle (three vertices)
triStream.Append(gout[1]);   ->m0
triStream.Append(gout[5]);   ->P1
triStream.Append(gout[3]);   ->m1
}
[maxvertexcount(8)]
void GS(triangle VertexOut gin[3], inout TriangleStream<GeoOut>)
{
VertexOut v[6];
Subdivide(gin, v);
OutputSubdivision(v, triStream);
}
```
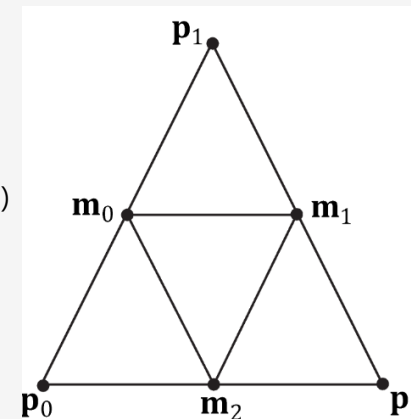
Binding GS to the rendering pipeline

Geometry shaders are compiled very similarly to vertex and pixel shaders. Suppose we have a geometry shader called GS in *TreeSprite.hlsl*, then we would compile the shader to bytecode like so:

```
mShaders["treeSpritePS"] =
d3dUtil::CompileShader(L"Shaders\\TreeSpr
ite.hlsl", alphaTestDefines, "PS",
"ps_5_0");
```

Like vertex and pixel shaders, a given geometry shader is bound to the rendering pipeline as part of a pipeline state object (PSO):

```cpp
// PSO for tree sprites - void TreeBillboardsApp::BuildPSOs()
D3D12_GRAPHICS_PIPELINE_STATE_DESC treeSpritePsoDesc = opaquePsoDesc;
treeSpritePsoDesc.VS =
{
reinterpret_cast<BYTE*>(mShaders["treeSpriteVS"]->GetBufferPointer()),
mShaders["treeSpriteVS"]->GetBufferSize()
};
treeSpritePsoDesc.GS =
{
reinterpret_cast<BYTE*>(mShaders["treeSpriteGS"]->GetBufferPointer()),
mShaders["treeSpriteGS"]->GetBufferSize()
};
treeSpritePsoDesc.PS =
{
reinterpret_cast<BYTE*>(mShaders["treeSpritePS"]->GetBufferPointer()),
mShaders["treeSpritePS"]->GetBufferSize()
};
treeSpritePsoDesc.PrimitiveTopologyType = D3D12_PRIMITIVE_TOPOLOGY_TYPE_POINT;
treeSpritePsoDesc.InputLayout = { mTreeSpriteInputLayout.data(),
(UINT)mTreeSpriteInputLayout.size() };
treeSpritePsoDesc.RasterizerState.CullMode = D3D12_CULL_MODE_NONE;

ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(&treeSpritePsoDesc,
IID_PPV_ARGS(&mPSOs["treeSprites"])));

// void TreeBillboardsApp::Draw(const GameTimer& gt)

mCommandList->SetPipelineState(mPSOs["treeSprites"].Get());
DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)RenderLayer::AlphaTestedTreeSprites]);
```
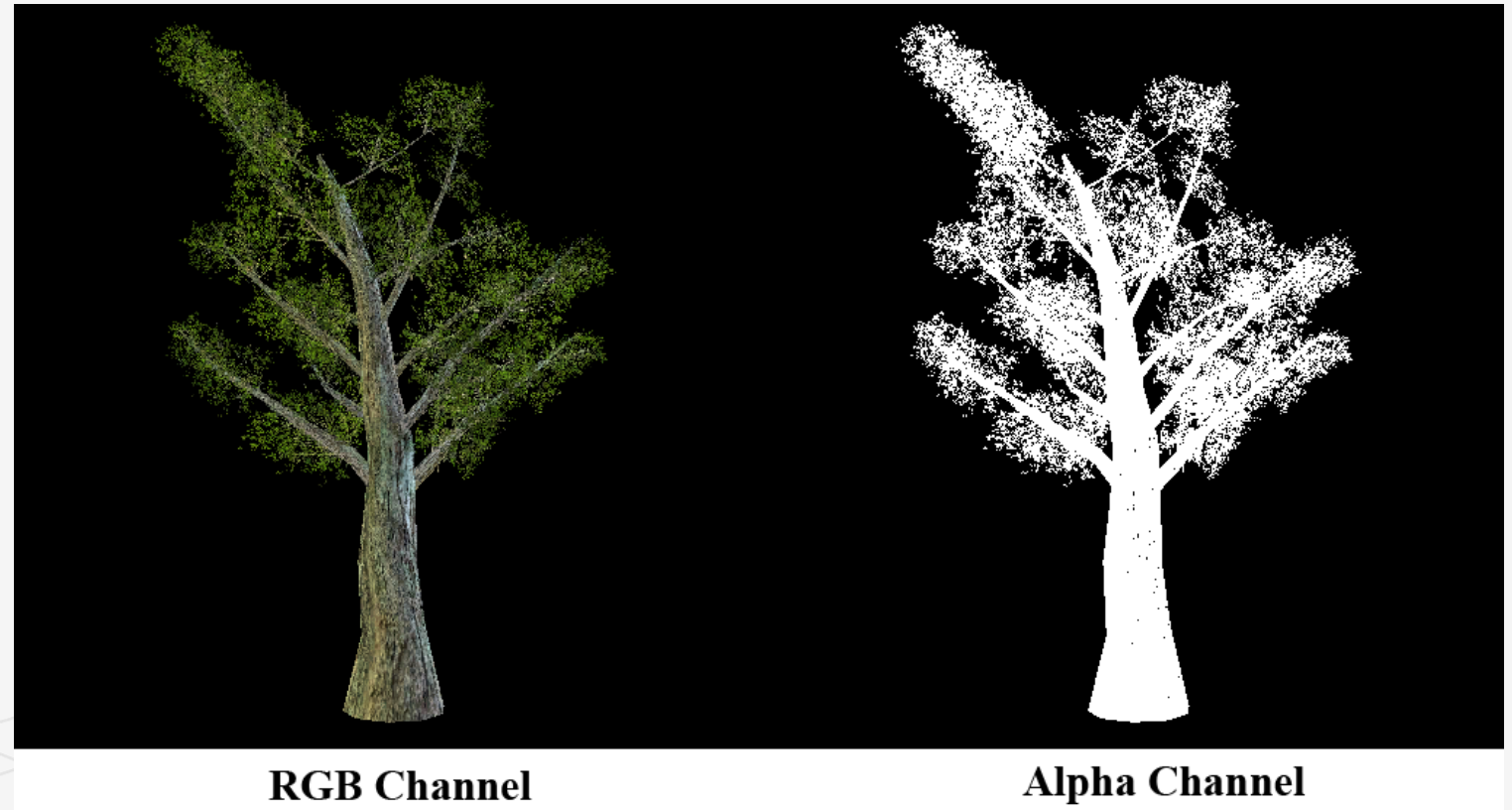
Billboards

When trees are far away, a billboarding technique is used for efficiency.

Instead of rendering the geometry for a fully 3D tree, a quad with a picture of a 3D tree is painted on it.

The trick is to make sure that the billboard always faces the camera.
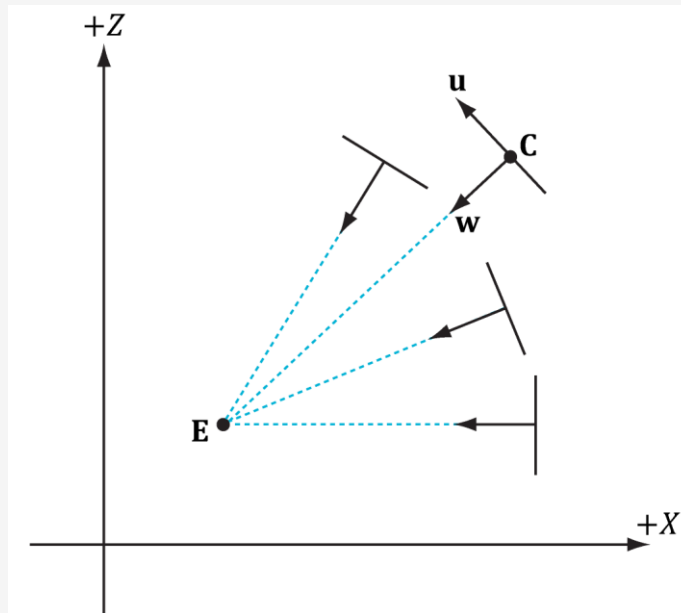
A tree billboard texture with alpha channel:

RGB Channel   **Alpha Channel**

Billboards

Assuming the *y*-axis is up and the *xz*-plane is the ground plane, the tree billboards will generally be aligned with the *y*-axis and just face the camera in the *xz*-plane.

Figure shows the local coordinate systems of several billboards from a bird's eye view—notice that the billboards are "looking" at the camera.

So given the center position **C** = (C_x, C_y, C_z) of a billboard in world space and the position of the camera **E** = (E_x, E_y, E_z) in world space, we have enough information to describe the local coordinate system of the billboard relative to the world space:

$$\mathbf{w} = \frac{\left(E_x - C_x, 0, E_z - C_z\right)}{\left\|\left(E_x - C_x, 0, E_z - C_z\right)\right\|}$$

$$\mathbf{v} = \left(0, 1, 0\right)$$

$$\mathbf{u} = \mathbf{v} \times \mathbf{w}$$

GS for our Billboard tree

Given the local coordinate system of the billboard relative to the world space, and the world size of the billboard, the billboard quad vertices can be obtained:

```
float3 up = float3(0.0f, 1.0f, 0.0f);

float3 look = gEyePosW - gin[0].CenterW;

look.y = 0.0f; // y-axis aligned, so project to xz-plane

look = normalize(look);

float3 right = cross(up, look);

// Compute triangle strip vertices (quad) in world space.

float halfWidth  = 0.5f*gin[0].SizeW.x;

float halfHeight = 0.5f*gin[0].SizeW.y;

float4 v[4];

v[0] = float4(gin[0].CenterW + halfWidth*right - halfHeight*up, 1.0f);

v[1] = float4(gin[0].CenterW + halfWidth*right + halfHeight*up, 1.0f);

v[2] = float4(gin[0].CenterW - halfWidth*right - halfHeight*up, 1.0f);

v[3] = float4(gin[0].CenterW - halfWidth*right + halfHeight*up, 1.0f);
```
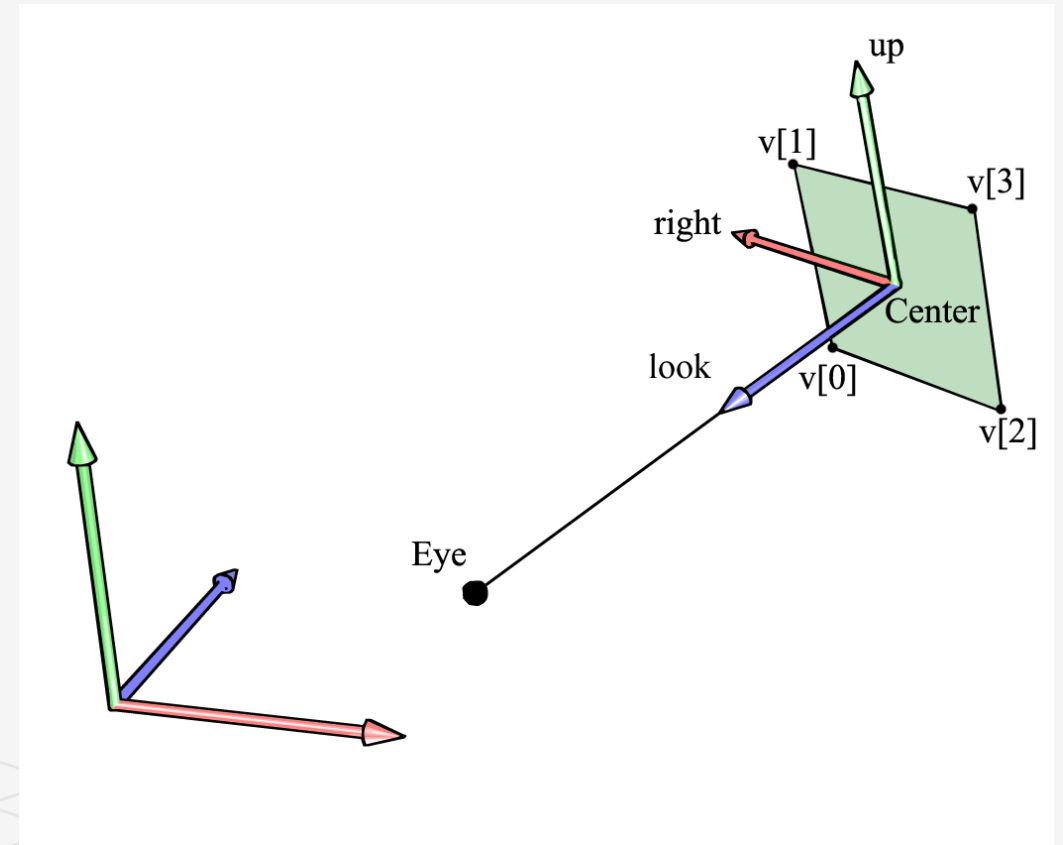
Tree Billboards

For PrimitiveTopologyType of the tree PSO, instead of D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE, we use D3D12_PRIMITIVE_TOPOLOGY_TYPE_POINT to construct a list of point primitives that lie above the mass land which represent the centers of the billboard.

```
treeSpritePsoDesc.PrimitiveTopologyType =
D3D12_PRIMITIVE_TOPOLOGY_TYPE_POINT;
```
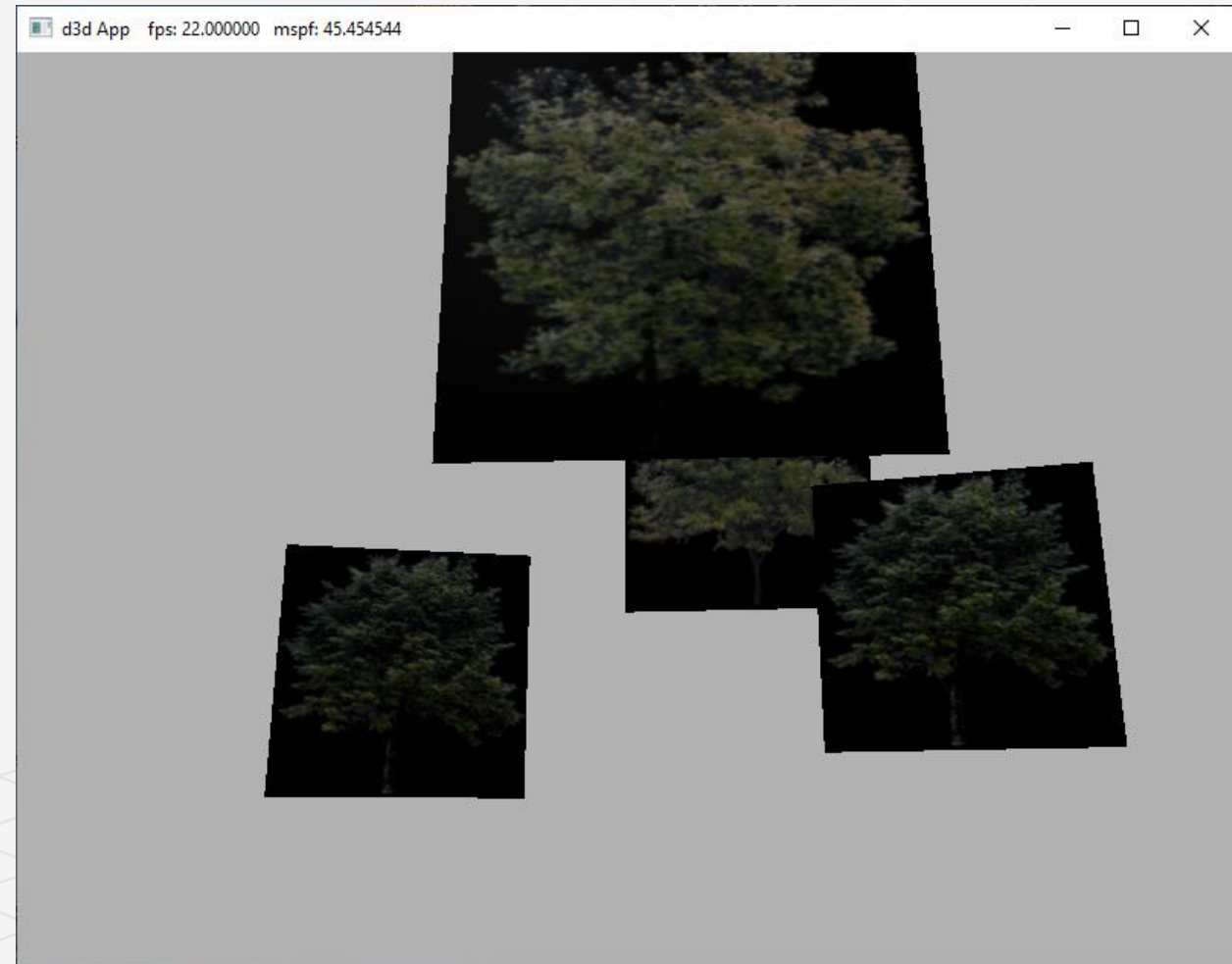
When we build our RenderITem:

```
treeSpritesRitem->PrimitiveType =
D3D_PRIMITIVE_TOPOLOGY_POINTLIST;
```

Which it's been set by:

```
cmdList->IASetPrimitiveTopology(ri->PrimitiveType);
```

In the geometry shader, we will expand these points into billboard quads.

Vertex Structure

The vertex stores a point which represents the center position of the billboard in world space.

The size member stores the width/height of the billboard (scaled to world space units); By having the size vary per vertex, we can easily allow for billboards of different sizes.

The geometry shader knows how large the billboard should be after expansion.

The figure shows how to expand a point to a quad.

```cpp
void TreeBillboardsApp::BuildTreeSpritesGeometry()
{
struct TreeSpriteVertex
{
XMFLOAT3 Pos;
XMFLOAT2 Size;
};
…..
void TreeBillboardsApp::BuildShadersAndInputLayouts()
{
mTreeSpriteInputLayout =
{
{ "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },
{ "SIZE", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 12,
D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },
};
```
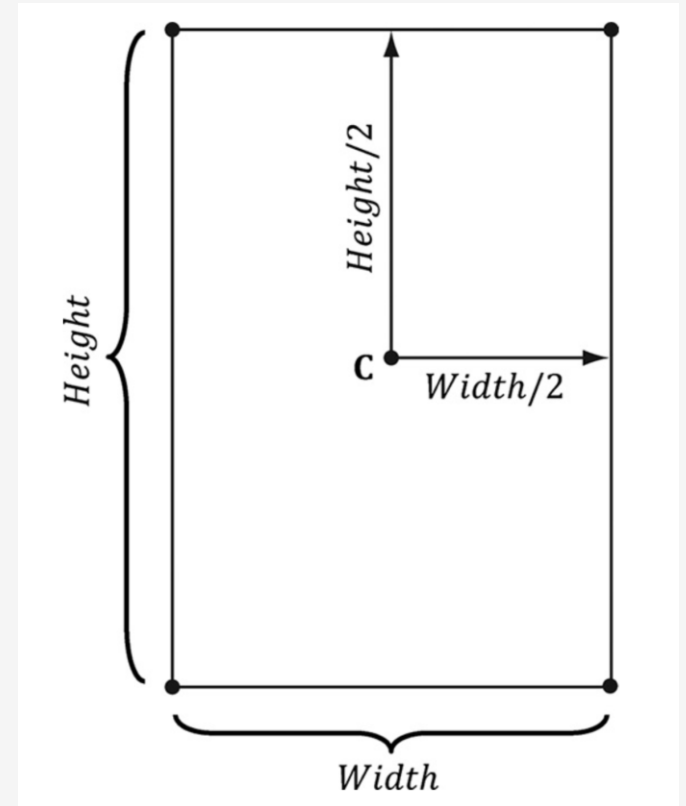
TreeBillboardsApp::BuildTreeSpritesGeometry()

```cpp
void TreeBillboardsApp::BuildTreeSpritesGeometry()
{
//step5
struct TreeSpriteVertex
{
XMFLOAT3 Pos;
XMFLOAT2 Size;
};


static const int treeCount = 4;

std::array<TreeSpriteVertex, 4> vertices;
for(UINT i = 0; i < treeCount; ++i)
{
float x = MathHelper::RandF(-45.0f, 45.0f);
float z = MathHelper::RandF(-45.0f, 45.0f);
float y = GetHillsHeight(x, z);

// Move tree slightly above land height.
y += 8.0f;

vertices[i].Pos = XMFLOAT3(x, y, z);
vertices[i].Size = XMFLOAT2(20.0f, 20.0f);
}
```

```cpp
std::array<std::uint16_t, 4> indices =
{
0, 1, 2, 3,
};

const UINT vbByteSize = (UINT)vertices.size() * sizeof(TreeSpriteVertex);
const UINT ibByteSize = (UINT)indices.size() * sizeof(std::uint16_t);

auto geo = std::make_unique<MeshGeometry>();
geo->Name = "treeSpritesGeo";

ThrowIfFailed(D3DCreateBlob(vbByteSize, &geo->VertexBufferCPU));
CopyMemory(geo->VertexBufferCPU->GetBufferPointer(), vertices.data(), vbByteSize);

ThrowIfFailed(D3DCreateBlob(ibByteSize, &geo->IndexBufferCPU));
CopyMemory(geo->IndexBufferCPU->GetBufferPointer(), indices.data(), ibByteSize);

geo->VertexBufferGPU = d3dUtil::CreateDefaultBuffer(md3dDevice.Get(),
mCommandList.Get(), vertices.data(), vbByteSize, geo->VertexBufferUploader);

geo->IndexBufferGPU = d3dUtil::CreateDefaultBuffer(md3dDevice.Get(),
mCommandList.Get(), indices.data(), ibByteSize, geo->IndexBufferUploader);

geo->VertexByteStride = sizeof(TreeSpriteVertex);
geo->VertexBufferByteSize = vbByteSize;
geo->IndexFormat = DXGI_FORMAT_R16_UINT;
geo->IndexBufferByteSize = ibByteSize;

SubmeshGeometry submesh;
submesh.IndexCount = (UINT)indices.size();
submesh.StartIndexLocation = 0;
submesh.BaseVertexLocation = 0;

geo->DrawArgs["points"] = submesh;

mGeometries["treeSpritesGeo"] = std::move(geo);
}
```

Alpha Test

In the pixel shader, we grab the alpha component of the texture. If it is a small value close to 0, which indicates that the pixel is completely transparent, then we clip the pixel from further processing.

This helps us to get rid of the black background!

```
const D3D_SHADER_MACRO alphaTestDefines[]
={"ALPHA_TEST"};

mShaders["treeSpritePS"] =
d3dUtil::CompileShader(L"Shaders\\TreeSpr
ite.hlsl", alphaTestDefines, "PS",
"ps_5_1");
```
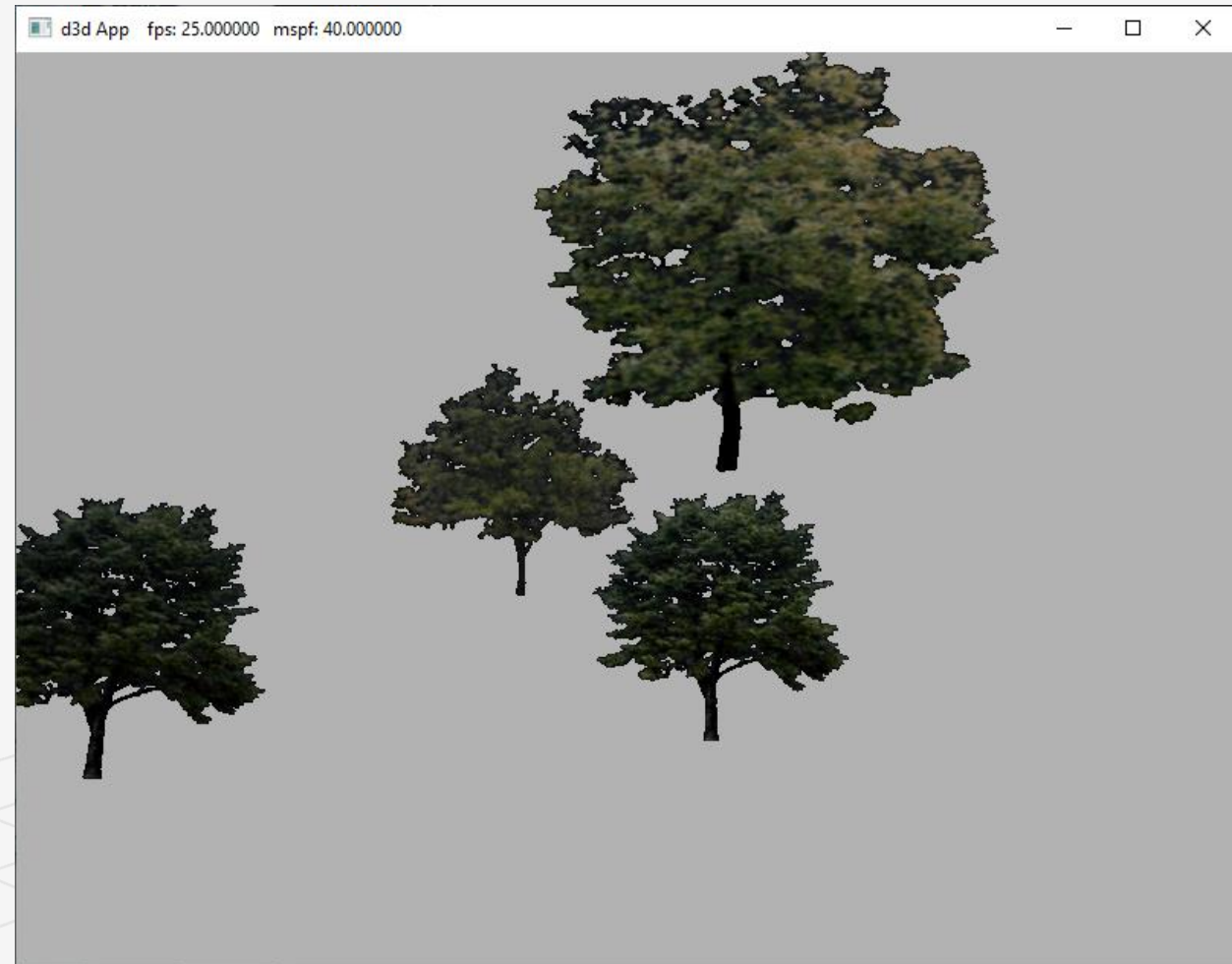
The HLSL File

The geometry shader takes a special unsigned integer parameter with semantic **SV_PrimitiveID**

```
[maxvertexcount(4)]

void GS(point VertexOut gin[1],

    uint primID : SV_PrimitiveID,

    inout TriangleStream<GeoOut> triStream)
```

When this semantic is specified, it tells the input assembler stage to automatically generate a primitive ID for each primitive.

When a draw call is executed to draw *n* primitives, the first primitive is labeled 0; the second primitive is labeled 1; and so on.

The primitive IDs are only unique for a single draw call.

In our billboard example, the geometry shader does not use this ID, instead, the geometry shader writes the primitive ID to the outgoing vertices, thereby passing it on to the pixel shader stage.

The pixel shader uses the primitive ID to index into a texture array.

- *If a geometry shader is not present, the primitive ID parameter can be added to the parameter list of the pixel shader:* float4 PS(VertexOut pin, uint primID : SV_PrimitiveID) : SV_Target

- *It is also possible to have the input assembler generate a vertex ID. To do this, add an additional parameter of type* uint *to the vertex shader signature with semantic* SV_VertexID: VertexOut VS(VertexIn vin, uint vertID : SV_VertexID)

TEXTURE ARRAYS

A texture array stores an array of textures.

In C++ code, a texture array is represented by the ID3D12Resource interface.

ID3D12Resource has a property called DepthOrArraySize that can be set to specify the number of texture elements the texture stores.

Look at the CreateD3DResources12 function in Common/DDSTextureLoader.cpp!

In a HLSL file, a texture array is represented by the Texture2DArray type:

```
Texture2DArray gTreeMapArray :
register(t0);
```

Why Texture2DArray, why not just do like this:

```
Texture2D TexArray[4];
```

In shader model 5.1 (new to Direct3D 12), we actually can do this. However, this was not allowed in previous Direct3D verrsions.

```
float4 PS(GeoOut pin) : SV_Target

{

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

Indexing textures like this may have a little overhead depending on the hardware.

Sampling a Texture Array

we sample a texture array with the following code:

When using a texture array, three texture coordinates are required.

The first two texture coordinates are the usual 2D texture coordinates;

the third texture coordinate is an index into the texture array. For example, 0 is the index to the first texture in the array, 1 is the index to the second texture in the array, 2 is the index to the third texture in the array, and so on.

```
float4 PS(GeoOut pin) : SV_Target

{

float3 uvw = float3(pin.TexC, pin.PrimID%3);

float4 diffuseAlbedo = gTreeMapArray.Sample(gsamAnisotropicWrap, uvw) * gDiffuseAlbedo;
```

In the Billboards demo, we use a texture array with four texture elements, each with a different tree texture.

However, because we are drawing more than three trees per draw call, the primitive IDs will become greater than two. Thus, we take the primitive ID modulo 3 (pin.PrimID % 3) to map the primitive ID to 0, 1, or 2 which are valid array indices for an array with four elements.

Texture Arrays

One of the advantages with texture arrays is that we were able to draw a collection of primitives, with different textures, in one draw call. Normally, we would have to have a separate render-item for each mesh with a different texture:

SetTextureA();

DrawPrimitivesWithTextureA();

SetTextureB();

DrawPrimitivesWithTextureB();

SetTextureZ();

DrawPrimitivesWithTextureZ();

Each set and draw call has some overhead associated with it. With texture arrays, we could reduce this to one set and one draw call:

SetTextureArray();

DrawPrimitivesWithTextureArray();

DIRECTX TEXTURE LIBRARY (DirectXTex)

https://walbourn.github.io/directxtex/

DirectXTex is a shared source library for reading and writing DDS files, and performing various texture content processing operations including resizing, format conversion, mip-map generation, block compression for Direct3D runtime texture resources, and height-map to normal-map conversion.

Texconv: This DirectXTex sample is an implementation of the "texconv" command-line texture utility   from the DirectX SDK utilizing DirectXTex rather than D3DX.
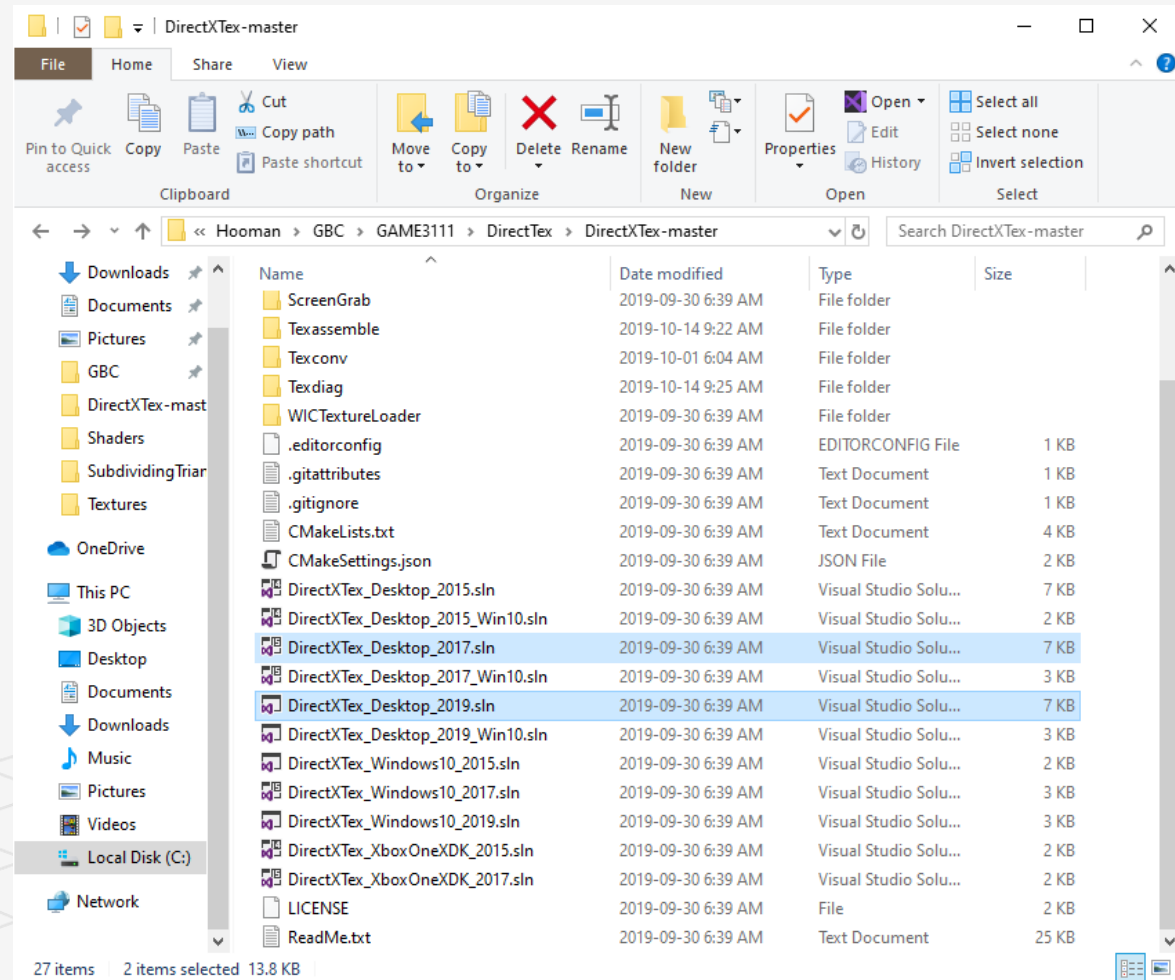
Texassemble: This DirectXTex sample is a command-line utility for creating cubemaps, volume maps, or   texture arrays from a set of individual input image files.

Download the DirectXTex from githu, open and build the DirectXTex_Desktop_2017.sln or DirectXTex_Desktop_2019.sln

This will create the utilities under

\DirectXTex-master\Texassemble\Bin\Desktop_2017\Win32\Debug

\DirectXTex-master\Texassemble\Bin\Desktop_2019\Win32\Debug

Loading Texture Arrays

Our DDS loading code in

Common/DDSTextureLoader.h/.cpp supports loading

DDS files that store texture arrays. So the key is to

create a DDS file that contains a texture array. To do

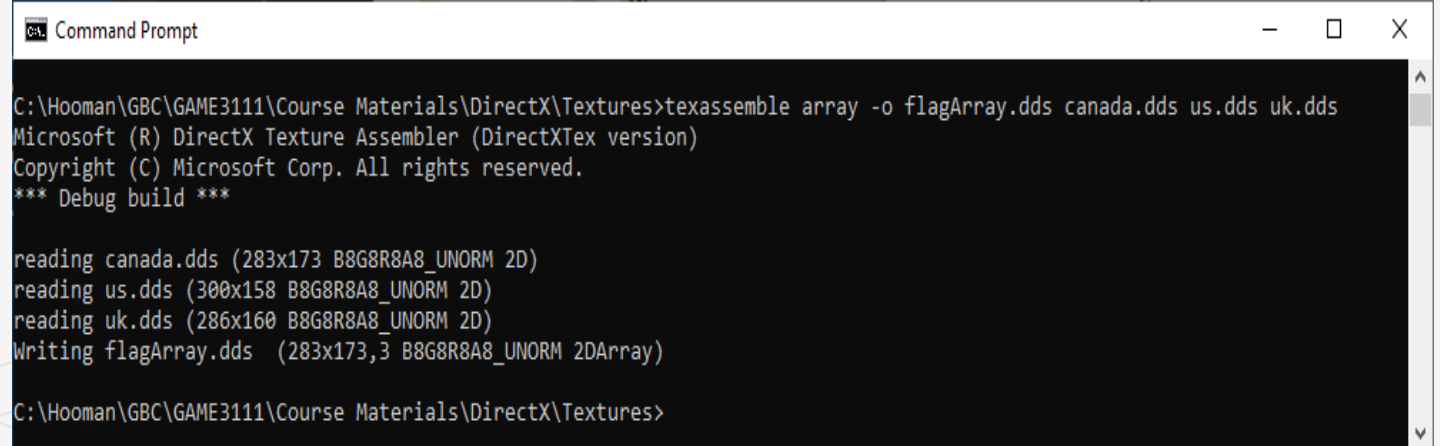this, we use the *texassemble* tool provided by

Microsoft.

The following syntax shows how to create a texture

array called *treeArray.dds* from 4 images *t0.dds, t1.dds,*

t2.dds, and *t3.dds*:

texassemble array -o treeArray.dds t0.dds t1.dds t2.dds

Note that when building a texture array with *texassemble*, the input images should only have one mipmap level. After you have invoked *texassemble* to build the texture array, you can use *texconv* togenerate mipmaps and change the pixel format if needed:

texconv -m 10 -f BC3_UNORM treeArray.dds

Or you can do the same by open the dds file using visual studio, and use the advance button to generate mipmap levels.

```
Command Prompt                                       —   □   X

C:\Hooman\GBC\GAME3111\Course Materials\DirectX\Textures>texassemble array -o flagArray.dds canada.dds us.dds uk.dds
Microsoft (R) DirectX Texture Assembler (DirectXTex version)
Copyright (C) Microsoft Corp. All rights reserved.
*** Debug build ***

reading canada.dds (283x173 B8G8R8A8_UNORM 2D)
reading us.dds (300x158 B8G8R8A8_UNORM 2D)
reading uk.dds (286x160 B8G8R8A8_UNORM 2D)
Writing flagArray.dds  (283x173,3 B8G8R8A8_UNORM 2DArray)

C:\Hooman\GBC\GAME3111\Course Materials\DirectX\Textures>
```
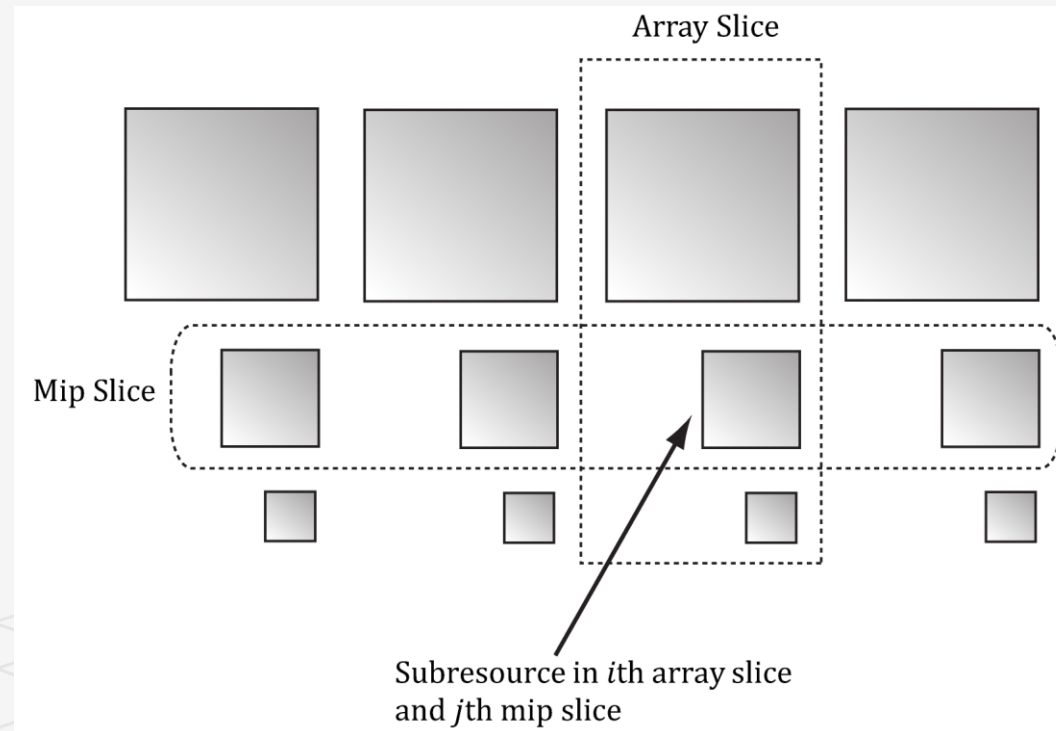
Texture Subresources

Figure shows an example of a texture array with several textures. In turn, each texture has its own mipmap chain.

The Direct3D API uses the term *array slice* to refer to an element in a texture along with its complete mipmap chain.

The Direct3D API uses the term *mip slice* to refer to all the mipmaps at a particular level in the texture array.

A subresource refers to a single mipmap level in a texture array element.

A texture array with four textures. Each texture has three mipmap levels.

Texture Subresources

Given the texture array index, and a mipmap level, we can access a subresource in a texture array. However, the subresources can also be labeled by a linear index; Direct3D uses a linear index ordered as shown in Figure.
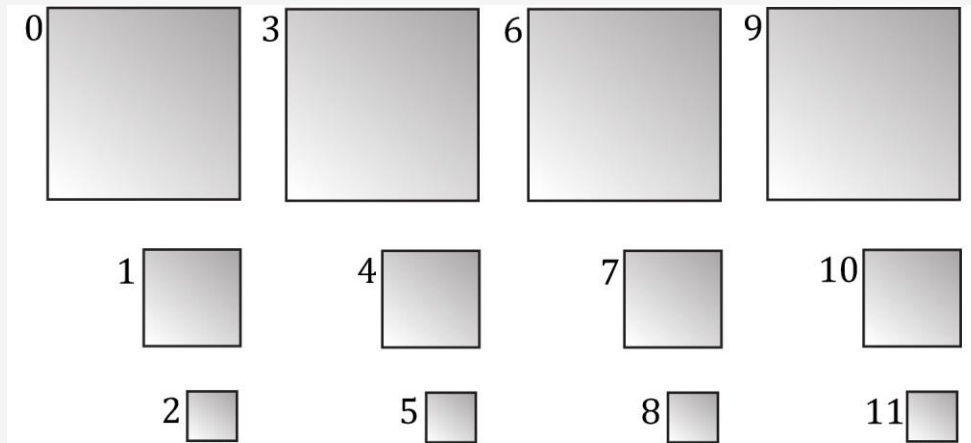
The following utility function is used to compute the linear subresource index given the mip level, array index, and the number of mipmap levels:

```
inline UINT D3D12CalcSubresource( UINT
MipSlice, UINT ArraySlice, UINT PlaneSlice,
UINT MipLevels, UINT ArraySize )

{

    return MipSlice + ArraySlice * MipLevels
+ PlaneSlice * MipLevels * ArraySize;

}
```

ALPHA-TO-COVERAGE

At some distances the edges of the tree billboard cutouts appear blocky.

This is caused by the clip function, which we use to mask out the pixels of the texture that are not part of the tree; the clip function either keeps a pixel or rejects it—there is no smooth transition.

The distance from the eye to the billboard plays a role because the short distances result in magnification, which makes the block artifacts larger, and short distances result in a lower resolution mipmap level being used.

MSAA executes the pixel shader once per pixel, at the pixel center, and then shares that color information with its subpixels based on visibility (the depth/stencil test is evaluated per subpixel) and coverage (does the subpixel center lie inside or outside the polygon?).

The key here is that *coverage is determined at the polygon level*. Therefore, MSAA is not going to detect the edges of the tree billboard cutouts as defined by the alpha channel.

When MSAA is enabled, and alpha-to-coverage is enabled (a member of D3D12_BLEND_DESC::AlphaToCoverageEnable = true), the hardware will look at the alpha value returned by the pixel shader and use that to determine coverage.

```
transparentPsoDesc.BlendState.AlphaToCoverageEnable = true;
```

For example, with 4X MSAA, if the pixel shader alpha is 0.5, then we can assume that two out of the four subpixels are covered and this will create a smooth edge.

The general advice is that you always want to use alpha-to-coverage for alpha masked cut out textures like foliage and fences. However, it does require that MSAA is enabled. Note that in the constructor of our demo application, we set:

m4xMsaaState = true;

This causes our sample framework to create the back and depth buffers with 4X MSAA support.