# 几何着色器及应用：树广告牌

# 概述

几何着色器（geometry shader）阶段是一个可选阶段，它位于顶点着色器和像素着色器阶段之间。

几何着色器以完整的图元作为输入数据，输出则为图元列表。

几何着色器的主要优点就是它可以创建或销毁几何体，例如，几何着色器可以将输入图元扩展为一个或多个其他图元，或者根据一些条件屏蔽某些图元的输出。

几何着色器的输出图元由一个顶点列表来描述。在顶点离开几何着色器之前，顶点坐标必须变换到齐次裁剪空间。在几何着色器阶段之后，顶点列表描述的是齐次裁剪空间中的图元。与往常一样，这些顶点会被投影（齐次除法），随后进行光栅化处理。

# 格式

|  |
| --- |
| [maxvertexcount(N)]  **void** ShaderName (  PrimitiveType InputVertexType InputName [NumElements],  inout StreamOutputObject<OutputVertexType>OutputName)  {  // Geometry shader body...  } |

[maxvertexcount(N)] 指定每次调用几何着色器时所能输出的顶点的最大数量。从性能方面考虑，maxvertexcount应竟可能小。

输入参数总是一个顶点数组，它可以表示：单个顶点、由两个顶点构成的直线、由3个顶点构成的三角形、由4个顶点构成的带有邻接信息的直线、由6个顶点构成的带有邻接信息的三角形。

输入的顶点类型（InputVertexType）与顶点着色器返回的顶点类型相同。

输入参数必须加上一个图元类型前缀，描述将要输入到几何着色器的图元类型（PrimitiveType）。可以使用的图元类型包括：

1．**point**：输入图元为点。

2．**line**：输入图元为直线（列表或线带）。

3．**triangle**：输入图元为三角形（列表或线带）。

4．**lineadj**：输入图元为带有邻接信息的直线（列表或线带）。

5．**triangleadj**：输入图元为带有邻接信息的三角形（列表或线带）。

注意：几何着色器的输入图元总是一个完整的图元（例如，由两个顶点构成一条直线、由三个顶点构成一个三角形）。这样，几何着色器就不需要区分列表和线带了。例如，在绘制三角形线带时，几何着色器会处理线带中的每个三角形，而且每个三角形的3个顶点都会作为输入数据传入到几何着色器中。这会导致额外的工作，因为几何着色器会重复处理被多个图元共享的顶点。

输出参数总是带有**inout**修饰符，并且是一个流类型对象。流类型用于存储由几何着色器输出的几何体顶点列表。

几何着色器使用内置的**Append**方法向输出流添加顶点。

流类型是一种模板类型， 其中的模板参数用于指定输出顶点的类型。这里有3种可以使用的流类型：

1．**PointStream**<OutputVertexType>：描述单个点的顶点列表。

2．**LineStream**<OutputVertexType>：描述直线线带的顶点列表。

3．**TriangleStream**<OutputVertexType>：描述三角形线带的顶点列表。

几何着色器以图元为单位输出顶点；输出图元的类型由流类型决定。对于直线和三角形来说，输出图元总是一个线带。

我们可使用内置的**RestartStrip**方法来创建新的线带从而模拟输出直线列表和三角形列表。

给定一个输入图元，几何着色器可以不对它进行输出。通过这一方式，几何着色器可以将输入的几何体“销毁”，这一功能在某些算法中非常有用。

当几何着色器输出的顶点无法构成一个完整的图元时，这部分图元将被丢弃。

# 例子

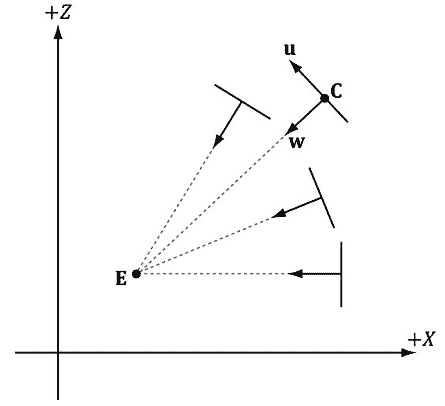
将一个三角形细分为4个三角形

|  |
| --- |
| struct VertexOut  {  float3 posL : POSITION;  float3 normalL : NORMAL;  float2 Tex : TEXCOORD;  };  struct GeoOut  {  loat4 posH : SV\_POSITION;  float3 posW : POSITION;  float3 normalW : NORMAL;  float2 Tex : TEXCOORD;  **float** FogLerp : FOG;  }    **void** Subdivide (VertexOut inVerts[3], out VertexOut outVerts[6])  {  // 1  // \*  // / \  // / \  // m0\*-----\*m1  // / \ /\  // / \ / \  // \*-----\*-----\*  //0 m2 2    VertexOut m[3];  // 计算每条边的中点  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);  // 投影到一个单位圆上  m[0].PosL = normalize(m[0].PosL);  m[1].PosL = normalize(m[1].PosL);  m[2].PosL = normalize(m[2].PosL);  // 求得法线  m[0].NormalL = m[0]. Pos L;  m[1].NormalL = m[1]. Pos L;  m[2].NormalL = m[2]. Pos L;  // 插值求得纹理坐标  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)  {  // 转换到世界空间  gout[i].PosW = mul(float4(v[i].PosL,1.0f), gWorld).xyz;  gout[i].NormalW = mul(v[i].NormalL,(float3x3)gWorldInvT ranspose);  // 转换到齐次剪裁空间  gout[i].PosH = mul(float4(v[i].PosL,1.0f), gWorldViewProj);  gout[i].Tex = v[i].Tex;  }  // 1  // \*  // / \  // / \  // m0\*-----\*m1  // / \ /\  // / \ / \  // \*-----\*-----\*  //0 m2 2    // 我们可以使用两个线带绘制细分三角形:  // 第一个：底部的三个三角形  // 第二个:顶部的一个三角形  [unroll]  **for**(**int** j = 0; j < 5; ++j )  {  triStream.Append(gout[j]);  }  triStream.RestartStrip();  triStream.Append(gout[1]);  triStream.Append(gout[5]);  triStream.Append(gout[3]);  }    [maxvertexcount(8)]  **void** GS(triangle VertexOut gin[3], inout TriangleStream<GeoOut>)  {  VertexOut v[6];  Subdivide(gin,v);  OutputSubdivision(v, triStream);  } |

# 树广告牌

当树与观察点的距离很远时，我们可以通过广告牌（billboard）技术来提高渲染效率。也就是，我们只在一个四边形上绘制树的3D图片，而不是渲染一个完整的3D树模型。从远处看，你根本分辨不出是否使用了广告牌。不过，你必须确保广告牌始终面对摄像机（否则个假象就会被拆穿）。

假设y轴垂直向上，xz平面为地平面。树广告牌总与y轴对齐，只在xz平面上面对摄像机。图11.3展示了鸟瞰视图中的几个广告牌的局部坐标系——注意，广告牌始终“面对”摄像机。



我们将创建一个点图元列表（D3D11\_PRIMITIVE\_TOPOLOGY\_POINTLIST），这些点描述了我们所要绘制的广告牌的中心位置。在几何着色器中，我们将这些点扩展为广告牌四边形，并计算广告牌的世界矩阵。

# Shader示例

|  |
| --- |
| // We expand each point into a quad (4 vertices), so the maximum number of vertices  // we output per geometry shader invocation is 4.  [maxvertexcount(4)]  **void** GS(point VertexOut gin[1],  uint primID : SV\_PrimitiveID,  inout TriangleStream<GeoOut> triStream)  {  //  // Compute the local coordinate system of the sprite relative to the world  // space such that the billboard is aligned with the y-axis and faces the eye.  //    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);    //  // Transform quad vertices to world space and output  // them as a triangle strip.  //    float2 texC[4] =  {  float2(0.0f, 1.0f),  float2(0.0f, 0.0f),  float2(1.0f, 1.0f),  float2(1.0f, 0.0f)  };    GeoOut gout;  [unroll]  **for**(**int** i = 0; i < 4; ++i)  {  gout.PosH = mul(v[i], gWorldViewProj);  gout.PosW = v[i].xyz;  gout.NormalW = look;  gout.TexC = texC[i];  gout.PrimID = primID;    triStream.Append(gout);  }  } |

在本例中，几何着色器包含了一个由**SV\_PrimitiveID**语义修饰的特殊的无符号整数参数。

当指定该语义时，输入汇编器阶段会为每个图元自动生成一个图元ID。当调用draw方法绘制n个图元时，第1个图元被标记为0，第2个图元被标记为1，依次类推，直至最后一个图元被标记为n−1。图元ID只有在每一次绘制调用中才是唯一的。

这里将该值传给了像素着色器阶段。像素着色器通过图元ID来建立广告牌和纹理数组之间的对应关系。

# 纹理数组

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）。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。



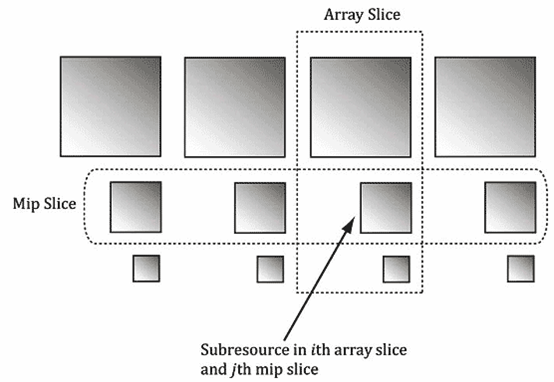
当对纹理数组进行采样时，我们需要3个纹理坐标。前两个纹理坐标是普通的2D纹理坐标；第3个纹理坐标是纹理数组的索引。

|  |
| --- |
| float3 uvw = float3(pin.Tex, pin.PrimID%3);  float4 diffuse = gTreeMapArray.Sample(samLinear,uvw); |

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，这样就避免了多次调用Draw的额外开销。

# 纹理子资源

下图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。



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按照下图所示的顺序标识线性索引。

