# 曲面细分阶段(Tessellation Stage)

# 概述

曲面细分技术是指将几何体细分为更小的三角形，从而丰富网格体的细节。在渲染流水线中，包含三个阶段:**Hull Shader**, **Tessellator Stage**, **Domian Shader**.

使用该技术有三点理由：

1.可以基于GPU实现动态LOD。可以根据摄像机与网格体的距离或者其他因素来调整细节。

2.可以简化物理模拟与动画。可以在低模(low-poly)网格体上执行物理与动画计算，然后使用曲面细分来获取更丰富的细节。

3.可以节约内存。我们在存储器里只保存低模，再根据需求来用GPU动态进行细分。

# 曲面细分的图元类型

在进行曲面细分时，我们不是向IA阶段提交三角形，而是提交含有若干控制点(control point)的面片(patch)。最多支持32个控制点。

例如需要细分4个点的矩形时，我们使用D3D11\_PRIMITIVE\_TOPOLOGY\_4\_CONTROL\_POINT\_PATCHLIST。

m\_d3dContext->IASetPrimitiveTopology(D3D11\_PRIMITIVE\_TOPOLOGY\_4\_CONTROL\_POINT\_PATCHLIST);

具有更多控制点的面片可以用贝塞尔曲线来生成贝塞尔曲面。

# 外壳着色器(Hull Shader)

该着色器实际上包含两个部分，常量外壳着色器和控制点外壳着色器。

**常量外壳着色器(Constant Hull Shader)**

其针对每个面片逐一处理，输出曲面细分因子。该因子决定了细分阶段中镶嵌化处理后得到的三角形数量。

例如下面是一个具有4个控制点的四边形面片，将其从各个方面均匀地分为3份(类似九宫格)。

|  |
| --- |
| struct PatchTess  {  **float** EdgeTess[4] : SV\_TessFactor;  **float** InsideTess[2] : SV\_InsideTessFactor;    // Additional info you want associated per patch.  };      PatchTess ConstantHS(InputPatch<VertexOut, 4> patch, uint patchID : SV\_PrimitiveID)  {  PatchTess pt;    // Uniformly tessellate the patch 3 times.  pt.EdgeTess[0] = 3; // Left edge  pt.EdgeTess[1] = 3; // Top edge  pt.EdgeTess[2] = 3; // Right edge  pt.EdgeTess[3] = 3; // Bottom edge    pt.InsideTess[0] = 3; // u-axis (columns)  pt.InsideTess[1] = 3; // v-axis (rows)    **return** pt;  } |

该着色器以所有控制点作为输入，使用**InputPatch<VertexOut, 4>**进行定义。其顶点类型即顶点着色器的输出类型，第二个参数则为控制点的数量。系统也通过**SV\_PrimitiveID**来获取面片的ID，可根据需要使用。

除了必须输出的**SV\_TessFactor**和**SV\_InsideTessFactor**外，我们也可以输出其他信息，以供之后的阶段使用。

对三角形面片的细分处理与四边形类似，需要3个边缘细分因子和一个内部细分因子。

Direct3D 11支持的细分因子最大为64。如果细分因子为0，则该面片会被丢弃。

出于性能考虑，有以下几点建议

1.如果曲面细分因子为1，就应考虑在渲染时不对其进行细分处理，否则会浪费资源。

2.对于过小的三角形，例如小于8个像素的，不宜进行细分处理。

3.使用细分时要采用批量绘制调用，将细分任务集中执行。反复开启/关闭细分功能的代价及其高昂。

**控制点外壳着色器(Control Point Hull Shader)**

该着色器以控制点作为输入和输出。其作用之一是改变曲面的表现方式。例如将普通的三角形(3个控制点)转换为3次贝塞尔三角曲面(具有10个控制点)。

下面是一个注简单的传递(pass-through)着色器，它不会对控制点进行任何修改。

|  |
| --- |
| struct HullOut  {  float3 PosL : POSITION;  };  [domain(“quad”)]  [partitioning(“integer”)]  [outputtopology(“triangle\_cw”)]  [outputcontrolpoints(4)]  [patchconstantfunc(“ConstantHS”)]  [maxtessfactor(64.0f)]  HullOut HS(InputPatch<VertexOut, 4> p, uint i : SV\_OutputControlPointID, uint patchId : SV\_PrimitiveID)  {  HullOut hout;  hout.PosL = p[i].PosL;  **return** hout;  } |

同样，InputPatch将所有控制点传入。**SV\_OutputControlPointID**则是当前输出控制点的ID。

**domain**:指定了面片的类型。可选有tri,quad,isoline。

**pattitioning**:指定了细分模式

    integer:表示新顶点的添加或移除只取决于细分因子的整数部分，忽略小数部分。这样在曲面细分等级改变时，容易发生明显的popping现象。

    非整型曲面细分(fractional\_even/fractional\_odd):细分因子的小数部分起到渐变过渡中调节的作用，使细分改变更加平滑。

**outputtopology**:细分创造的三角形的绕序。triangle\_cw表示顺时针，triangle\_ccw表示逆时针，line用于对线段进行细分。

**outputcontrolpoints**:外壳着色器执行的次数，每次执行都输出一个控制点，SV\_OutputControlPointID给出了当前正在输出的控制点ID。

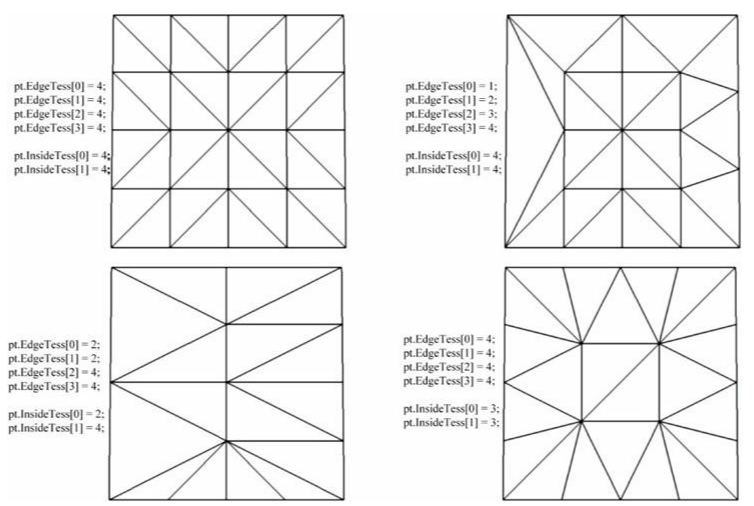
**patchconstantfunc**:指定常量外壳着色器的名称。

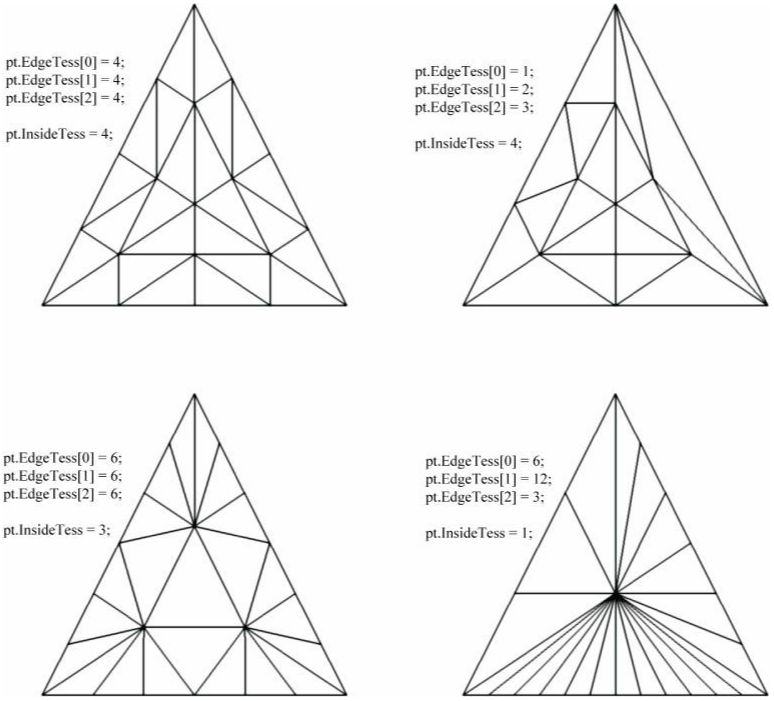
**maxtessfactor**:曲面细分因子的最大值，最大为64。硬件可针对此值进行相应的优化。

# 镶嵌器阶段(Tessellator Stage)

此阶段是硬件根据曲面细分因子执行细分处理的阶段。

几个示例





# 域着色器(Domian Shader)

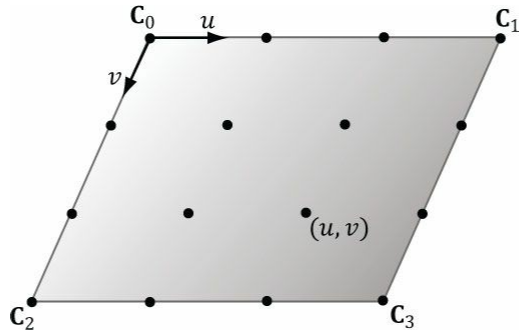
上个阶段创建出的所有顶点都会逐一调用该着色器。此着色器相当于没有开启曲面细分时的顶点着色器。我们可以在这里将顶点变换到齐次裁剪空间。

例子

|  |
| --- |
| struct DomainOut  {  float4 PosH : SV\_POSITION;  };      // The domain shader is called for every vertex created by the tessellator.  // It is like the vertex shader after tessellation.  [domain(“quad”)]  DomainOut DS(PatchTess patchTess, float2 uv : SV\_DomainLocation, **const** OutputPatch<HullOut, 4> quad)  {  DomainOut dout;    // Bilinear interpolation.  float3 v1 = lerp(quad[0].PosL, quad[1].PosL, uv.x);  float3 v2 = lerp(quad[2].PosL, quad[3].PosL, uv.x);  float3 p = lerp(v1, v2, uv.y);    dout.PosH = mul(float4(p, 1.0f), gWorldViewProj);  **return** dout;  } |

域着色器以曲面细分因子，控制点外壳着色器输出的所有面片，以及顶点坐标uv作为输入。

这里只是给出了uv坐标而不是实际位置。我们可以通过其求出实际的3D顶点位置。



处理三角形时，顶点坐标是float3型的重心坐标uvw。

# 演示程序:曲面细分实现山丘

我们向流水线提交一个四边形面片，然后根据摄像机与其之间的距离进行细分，然后再对顶点进行平移。

外壳着色器

|  |
| --- |
| struct PatchTess  {  **float** EdgeTess[4] : SV\_TessFactor;  **float** InsideTess[2] : SV\_InsideTessFactor;  };    PatchTess ConstantHS(InputPatch<VertexOut, 4> patch, uint patchID : SV\_PrimitiveID)  {  PatchTess pt;    float3 centerL = 0.25f\*(patch[0].PosL + patch[1].PosL + patch[2].PosL + patch[3].PosL);  float3 centerW = mul(float4(centerL, 1.0f), gWorld).xyz;    **float** d = distance(centerW, gEyePosW);    // Tessellate the patch based on distance from the eye such that  // the tessellation is 0 if d >= d1 and 64 if d <= d0. The interval  // [d0, d1] defines the range we tessellate in.    **const** **float** d0 = 20.0f;  **const** **float** d1 = 500.0f;  **float** tess = 64.0f\*saturate((d1 - d) / (d1 - d0));    // Uniformly tessellate the patch.    pt.EdgeTess[0] = tess;  pt.EdgeTess[1] = tess;  pt.EdgeTess[2] = tess;  pt.EdgeTess[3] = tess;    pt.InsideTess[0] = tess;  pt.InsideTess[1] = tess;    **return** pt;  } |

我们根据观察点与网格的距离执行saturate函数，使细分因子在0到64之间变化。

域着色器

|  |
| --- |
| struct DomainOut  {  float4 PosH : SV\_POSITION;  float3 PosW : POSITION;  float3 NormalW : NORMAL;  float2 TexUV : TEXUV;  };    // The domain shader is called for every vertex created by the tessellator.  // It is like the vertex shader after tessellation.  [domain("quad")]  DomainOut DS(PatchTess patchTess,  float2 uv : SV\_DomainLocation,  **const** OutputPatch<HullOut, 4> quad)  {  DomainOut dout;    // Bilinear interpolation.  float3 v1 = lerp(quad[0].PosL, quad[1].PosL, uv.x);  float3 v2 = lerp(quad[2].PosL, quad[3].PosL, uv.x);  float3 p = lerp(v1, v2, uv.y);    // Displacement mapping  p.y = 0.3f\*(p.z\*sin(0.1f\*p.x) + p.x\*cos(0.1f\*p.z));    dout.PosW = mul(float4(p, 1.0f), gWorld);  dout.PosH = mul(float4(p, 1.0f), gWorldViewProj);    // Calculate normal  float3 NormalL = float3(  -0.03f\*p.z\*cos(0.1f\*p.x) - 0.3f\*cos(0.1f\*p.z),  1.0f,  -0.3f\*sin(0.1f\*p.x) + 0.03f\*p.x\*sin(0.1f\*p.z));  NormalL = normalize(NormalL);  dout.NormalW = mul(NormalL, (float3x3)gWorldInvTranspose);    dout.TexUV = uv;    **return** dout;  } |

域着色器的输出需与像素着色器的输入相匹配。这里我们平移顶点，并给normal和uv赋值。