# OpenGL 4.0的Tessellation Shader（细分曲面着色器） - zenny_chen

https://www.cnblogs.com/zenny-chen/p/4280100.html

None Sun Oct, 11 19:24

细分曲面着色器（Tessellation Shader）处于顶点着色器阶段的下一个阶段，我们可以看以下链接的OpenGL渲染流水线的图：https://www.opengl.org/wiki/Rendering_Pipeline_Overview（可能需要翻墙）。它是由ATI在2001年率先设计出来的。

**细分曲面着色器**

直到这个阶段，对于操作几何图元而言，只有顶点着色器对我们可用。尽管使用顶点着色器可以使用不少图形技术，不过顶点着色器也确实存在一些限制。一个就是它们在执行过程中无法创建额外的几何图形。它们仅仅更新与它们当前所处理的顶点相关的数据。而且，它们甚至无法访问在当前图元中其它顶点的数据。

为了解决那些问题，OpenGL流水线含有几个其它着色器阶段来打破这些限制。我们这里所要介绍的*细分曲面着色器*可以使用一个新的几何图元类型，称为**patch**（斑点、碎片），来生成一个三角形网格。

细分曲面着色器在OpenGL流水线中增添了两个着色器阶段来生成一个几何图元的网格。比起在使用顶点着色器时不得不指定所有线与三角形来形成自己的模型，在使用细分曲面时，一开始指定一个patch，它是一列排好序的顶点。当一个patch被渲染时，*细分曲面控制着色器*先执行，对patch顶点进行操作，并指定从patch中应该生成多少几何图形。细分曲面控制着色器是可选的，我们后面会看到，如果不用它的话需要使用哪些条件。在细分曲面控制着色器完成之后，第二个着色器——*细分曲面计算着色器*使用细分曲面坐标来放置所生成的顶点，并且将它们发送到光栅化器，或发送到几何着色器做进一步处理。

**细分曲面Patch**

细分曲面过程并不对OpenGL典型的几何图元（点、线和三角形）进行操作，而是使用一个新的图元（在OpenGL 4.0版本中新增的），称为patch。patch由流水线中所有活动的着色阶段处理。相比起来，其它图元类型仅仅被顶点、片段和几何着色器处理，而旁通细分曲面阶段。实际上，如果有任一细分曲面着色器是活跃的，那么传递任何其它几何类型会产生一个GL_INVALID_OPERATION错误。相反地，如果企图渲染一个patch而没有任何细分曲面着色器（明确地说是一个细分曲面计算着色器；我们会看到细分曲面控制着色器是可选的），那么将也会得到一个GL_INVALID_OPERATION错误。

patch仅仅是传入到OpenGL的一列顶点列表，该列表在处理期间保存它们的次序。当用细分曲面与patch进行渲染时，使用像**glDrawArrays()**这样的渲染命令，并指定从绑定的顶点缓存对象（VBO）将被读出的顶点的总数，然后为该绘制调用进行处理。当用其它的OpenGL图元进行渲染时，OpenGL基于在绘制调用中所指定的图元类型而隐式地知道要使用多少顶点，比如使用三个顶点来绘制一个三角形。然后，当使用一个patch时，需要告诉OpenGL顶点数组中要使用多少个顶点来组成一个patch，而这可以通过使用**glPatchParameteri()**进行指定。由同一个绘制调用所处理的patch，它们的尺寸（即每个patch的顶点个数）将是相同的。

```
void glPatchParameteri(GLenum pname, GLint value);
/**
 * 使用value来指定一个patch中的顶点个数。pname必须设置为GL_PATCH_VERTICES。
 * 如果value小于零或大于GL_MAX_PATCH_VERTICES，将会产一个GL_INVALID_ENUM的错误。
 * 一个patch的默认顶点个数是三。如果一个patch的顶点个数小于参数value值，那么该patch将被忽略，从而不会有几何图形产生。
*/
```

要指定一个patch，使用类型GL_PATCHES输入到任一OpenGL绘制命令。以下代码描述了发射两个patch，每个patch含有四个顶点，然后通过glDrawArrays绘制命令进行渲染。

```
GLfloat vertices[][2] = {
    {-0.75f, -0.25f}, {-0.25f, -0.25f}, {-0.25f, 0.25f}, {-0.75f, 0.25f},
    {0.25f, -0.25f}, {0.75f, -0.25f}, {0.75f, -0.25f}, {0.75f, 0.25f},
{0.25f, 0.25f}
};

glBindVertexArray(VAO);
glBindBuffer(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
GL_STATIC_DRAW);
glVertexAttribPointer(vPos, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
glPatchParameteri(GL_PATCH_VERTICES, 4);
glDrawArrays(GL_PATCHES, 0, 8);
```

每个patch的顶点首先由当前绑定的顶点着色器处理，然后用于初始化数组gl_in，这个变量在细分曲面控制着色器中被隐式地声明。gl_in中的元素个数与由**glPatchParameteri()**所指定的patch大小相同。在一个细分曲面着色器内部，变量gl_PatchVerticesIn提供了gl_in中的元素个数（就好比使用sizeof(gl_in) / sizeof(gl_in[0])进行查询）。

### 细分曲面控制着色器

一旦应用发射了一个patch，细分曲面控制着色器就会被调用（如果有所绑定的话）并且负责完成以下行动：

● 生成*细分曲面输出patch顶点*，传递到细分曲面计算着色器，以及更新任一每个顶点的，或每个patch的属性值，若有必要的话。

● 指定*细分曲面程度因子*，控制图元生成器的操作。这些是特殊的细分曲面控制着色器变量，称为gl_TessLevelInner和gl_TessLevelOuter，并在细分曲面控制着色器中隐式声明。

我们将依次讨论这些行动的每一个。

**生成输出patch顶点**

细分曲面控制器使用由应用所指定的顶点——这些顶点我们称为*输入patch顶点*（作为顶点着色器的输出）——来生成一组新的顶点，这些新的顶点为输出patch顶点。它们存放在细分曲面控制着色器的gl_out数组中。细分曲面控制着色器在产出输出patch顶点时，可以修改传递自应用的值（比如顶点属性），也可以创建或移除来自输入patch顶点中的顶点。

使用一个layout构造在细分曲面控制着色器中指定输出patch顶点的个数。下面语句描述了设置输出patch顶点的个数为16。

```
layout (vertices = 16) out;
```

layout指示符中的vertices参数所设置的值做了两件事情：它设置了输出patch顶点gl_out的大小；并且指定了细分曲面控制着色器将被执行多少次：对每个输出patch顶点执行一次。

为了确定正在处理哪个输出顶点，细分曲面控制着色器可以使用gl_InvocationID变量。该变量最经常被用作为gl_out数组的一个索引。当一个细分曲面控制着色器在执行时，它具有对所有patch顶点数据的访问，包括输入顶点和输出顶点。这可能会导致发射一次着色器调用，该调用需要使用来自另一个着色器调用的数据值，但是那个着色器调用尚未发生。细分曲面控制着色器可以使用GLSL的**barrier()**函数，该函数使得对于一个输入patch的所有控制着色器执行并等待所有这些着色器的执行到达那个函数的调用点，从而确保了可能要设置的所有数据值将被计算。

细分曲面控制着色器的一个普遍的习惯用法仅仅是将输入patch顶点输出到此着色器的外面。下面的例子描述了带有四个顶点的一个输出patch。

```
#version 410 core

layout (vertices = 4) out;

void main(void)
{
    gl_out[gl_InvocationID].gl_Position =
gl_in[gl_InvocationID].gl_Position;

    // 下面设置细分曲面程度
}
```

**细分曲面控制着色器变量**

gl_in数组实际上是一个结构体的数组，每个元素被定义为：

```
in gl_PerVertex {
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];

}gl_in[gl_PatchVerticesIn];
```

并且对于每个需要向下一个阶段传递的值（比如，向下传递到细分曲面计算着色器），需要进行相应地赋值。如上述代码片段所述，传递了gl_Position变量。

gl_out数组具有相同的结构体成员，不过数组大小与gl_in不同，它是由gl_PatchVerticesOut来指定的。而这个值则是在细分曲面控制器中的out这一layout限定符中设置。此外，以下标量值用于确定正在被着色的图元和输出顶点：

gl_InvocationID：当前细分曲面着色器的输出顶点的调用索引

gl_PrimitiveID：当前输入patch的图元索引

gl_PatchVerticesIn：输入patch中的顶点个数，它作为gl_in数组变量中的元素个数

gl_PatchVerticesOut：输出patch中的顶点个数，它作为gl_out数组变量中的元素个数

如果我们需要额外的基于每个顶点的属性值，或为输入或为输出，那么这需要在我们的细分曲面控制着色器中将它们声明为in或out数组。一个输入数组的大小需要与输入patch大小相同，或者可以被声明为缺省大小的，这样OpenGL将会为其所有值适当地分配空间。类似地，每个顶点的输出属性需要与输出patch中的顶点个数相一致，也可以为输出属性声明为缺省大小的。输出属性值将会被传递到细分曲面计算着色器，作为其输入属性值。

比如：

```
#version 410 core

layout (vertices = 4) out;

in vec4 vertexCoeffs1[4];    // 我们假定指定一个输入patch含有4个顶点
in vec4 vertexCoeffs2[];     // 这里使用缺省大小的数组变量，OpenGL将会自动为其分配大小

out vec2 vertexTexCoord1[4];    // 这里需要用上面out的layout (vertices)值一致
out vec2 vertexTexCoord2[];     // 这里使用缺省大小的数组变量，OpenGL将会自动为其分配大小

void main(void)
{
    // Do something here
}
```

上面给出的是基于逐个顶点的属性值，它们可以用gl_InvocationID作为索引，不过要注意的是，gl_InvocationID标识的是当前细分曲面着色器的***输出***顶点的调用索引。我们可以使用patch限定符来声明每个patch的输出变量。每个patch的变量不是以数组方式定义而是以普通单实例变量的方式来定义。当然，我们也可以将它们定义为数组。所有细分曲面控制着色器的调用看到的都是同一个patch变量。比如：

```
#version 410 core

patch out vec4 data;

layout (vertices = 4) out;

void main(void)
{
    // Do something here    data = vec4(1.0, 0.0, 0.0, 1.0);
}
```

这里，我们定义了一个标识符为data的patch输出变量。对于每次细分曲面控制着色器的调用，data的值都被写为vec4(1.0, 0.0, 0.0, 1.0)。因此，任一细分曲面控制着色器可以写基于每个patch的输出变量；实际上，所有细分曲面控制着色器的调用一般都将写到一个基于每个patch的变量。只要它们都写相同的值，那么一切都是良好的。

### 控制细分曲面

一个细分曲面控制着色器的另一个功能是指定对输出patch细分多少。然而我们还没详细地讨论细分曲面计算着色器，它们控制用于渲染的输出patch的类型，结果也就是细分曲面所发生的域。OpenGL支持三种细分曲面域：四边形，三角形，和等值线集合。这些通过细分曲面计算着色器中的**in**的**layout**进行指定。

细分曲面的数量通过指定两组值：内部和外部细分曲面程度来控制的。外部细分曲面的值控制域的周边是如何划分的，然后存放在一个隐式声明的名为gl_TessLevelOuter的含有四个元素的数组中。而内部细分曲面程度指定了域的内部如何进行划分，然后存放在一个名为gl_TessLevelInner的含有两个元素的数组中。所有细分曲面程度因子是浮点值，并且我们将会看到浮点值在细分曲面上以一个比特的效果。最后一点是，尽管隐式声明的细分曲面程度因子数组的维度是固定的，不过从那些数组所使用的值的个数依赖于细分曲面域的类型。下面来看看这两个OpenGL内建的细分曲面输出patch变量的声明：

```
patch out float gl_TessLevelOuter[4];
patch out float gl_TessLevelInner[2];
```

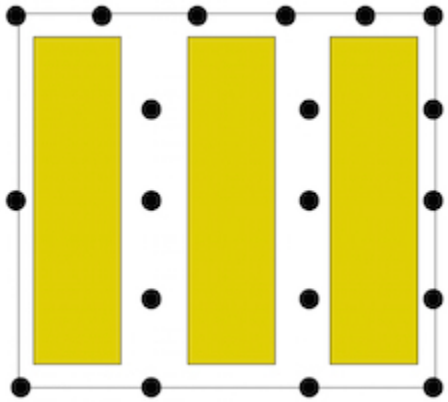
理解外部与内部细分曲面程度如何操作是让细分曲面做我们想要做的事情的关键。每个细分曲面程度因子指定了对一个区域划分多少条“线段”，以及生成多少细分曲面坐标与几何图元。这种划分如何完成根据不同域类型而有所不同。我们将依次讨论域的每种类型。

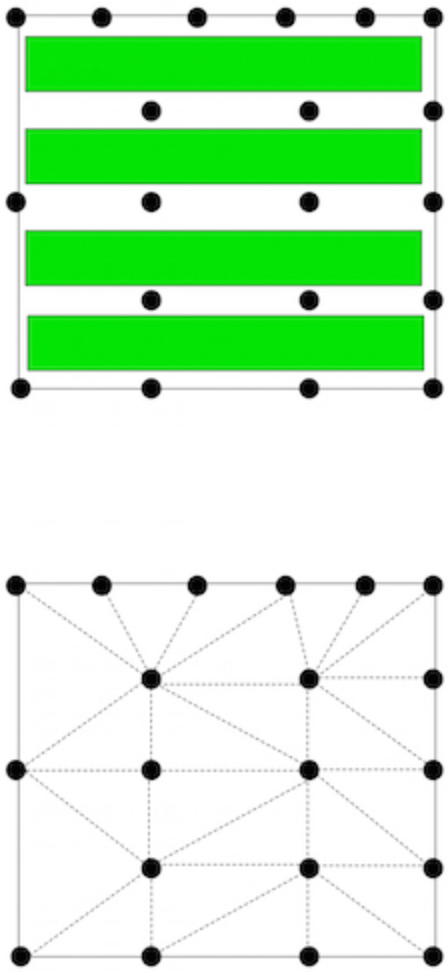
### 四边形细分曲面

使用四边形域可能是最直观的，因此我们先介绍这种类型。当输入patch形状为矩形时，这很有用，当我们可能使用二维的样条曲面时，比如Bézier曲面。四边形域使用所有内部和外部细分曲面程度来划分*单位正方形*。比如，如果我们用以下代码来设置细分曲面程度因子，那么OpenGL将会把四边形域细分为如下图所示的样子。

```
#version 410 core

layout (vertices = 4) out;

void main(void)
{
    gl_out[gl_InvocationID].gl_Position =
gl_in[gl_InvocationID].gl_Position;

    // 下面设置细分曲面程度
    gl_TessLevelInner[0] = 3.0;      // 内部划分3条垂直区域，即内部新增2列顶点
    gl_TessLevelInner[1] = 4.0;      // 内部划分4条水平区域，即内部新增3行顶点

    gl_TessLevelOuter[0] = 2.0;      // 左边2条线段
    gl_TessLevelOuter[1] = 3.0;      // 下边3条线段
    gl_TessLevelOuter[2] = 4.0;      // 右边4条线段
    gl_TessLevelOuter[3] = 5.0;      // 上边5条线段
}
```

下图为上述细分曲面控制着色器可能会得到的几何图形样子。





上图中，三条黄色线条表示三条垂直区域；四条绿色线条表示四条水平区域。

注意，外部细分曲面程度值对应于围绕周边的每条边的线段个数，而内部细分曲面程度指定了域的内部空间中水平与垂直方向上有多少个“区域”。使用虚线则是将整个域用三角形进行划分。域的三角形划分是依赖于实现的。实心圆点表示细分曲面坐标，每个坐标都会提供给细分曲面计算着色器，作为其输入。在四边形域的情况下，细分曲面坐标有两个坐标值$(u, v)$，两者值的范围都在$[0, 1]$范围内，并且每个细分曲面坐标将传递到细分曲面计算着色器中，作为它的一次调用。

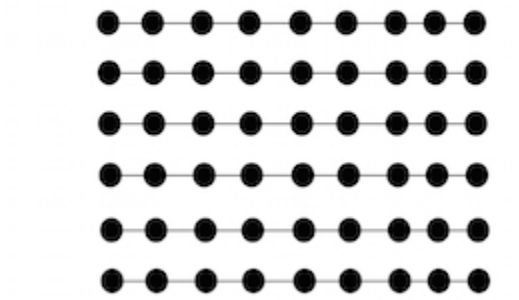
**等值线细分曲面**

类似于四边形域，等值线域也生成$(u, v)$对作为给细分曲面计算着色器的细分曲面坐标。然而，等值线仅仅使用外部细分曲面程度的两个元素值来判定划分量（这里没有用到内部细分曲面程度）。

下面是设置等值线域的细分曲面程度：

```
gl_TessLevelOuter[0] = 6;       // 6条等值线
gl_TessLevelOuter[1] = 8;       // 每条等值线被划分为8条线段
```

下图为可能的结果图形：



**旁通细分曲面控制着色器**

正如我们所提到过的，细分曲面着色器往往只是一个直通着色器，将数据从输入直接到输出。在这种情况下，我们实际上可以使一个细分曲面着色器进行旁通，仅仅使用主机端OpenGL API来设置细分曲面程度因子。使用**glPatchParameterfv()**函数来设置内部与外部细分曲面程度。

```
void glPatchParameterfv(GLenum pname, const GLfloat *values);

/**
 * 当没有细分曲面控制着色器时，设置内部与外部细分曲面程度。
 * 参数pname要么是GL_PATCH_DEFAULT_OUTER_LEVEL，要么是
GL_PATCH_DEFAULT_INNER_LEVEL。
 * 当pname是GL_PATCH_DEFAULT_OUTER_LEVEL时，参数values必须是含有四个单精度浮点值的
数组，指定四个外部细分曲面程度。
 * 类似地，当pname是GL_PATCH_DEFAULT_INNER_LEVEL时，values必须是含有两个单精度浮点
值的数组，指定两个内部细分曲面程度。
*/
```

**细分曲面图元生成**

图元生成是一个固定功能阶段，负责从输入patch来创建一组新的图元。此阶段仅当一个细分曲面计算着色器在当前程序或程序流水线中活动时才会执行。图元生成受以下因素影响：

• 细分曲面程度

• 被细分的顶点的空间划分，这个由细分曲面计算着色器阶段进行定义。它可以是equal_spacing、fractional_even_spacing或fractional_odd_spacing。

• 由后续细分曲面着色器所定义的图元类型，即triangles、quads或isolines。细分曲面计算着色器也可以迫使细分曲面的生成作为一系列的点，而不是三角形或线。这个可以通过point_mode来指定。

• 由后续细分曲面计算着色器所定义的图元生成次序，比如cw（顺时针）或ccw（逆时针）。

**抽象patch**

注意，图元生成不受细分曲面控制着色器中用户定义的输出影响（或当没有细分曲面控制着色器时，也不受顶点着色器的影响），不受细分曲面控制着色器的输出patch大小影响，也不受任一每个patch的细分曲面控制着色器输出的影响，而只受细分曲面程度的影响。细分曲面阶段的图元生成部分完全对实际的顶点坐标与其它patch数据是不可见的。

图元生成系统的目的是确定要生成多少个顶点，用哪个次序来生成它们，以及用哪种图元来构造它们。实际为这些顶点的每个顶点的数据，诸如位置、颜色等等，是通过细分曲面计算着色器来生成的，基于图元生成器所提供的信息。

因为这种对分，图元生成器对可以被认为是一个“抽象patch”的东东进行操作。它并不从细分曲面控制器的输出来看patch；它仅考虑一个抽象四边形、三角形或等值线块的细分曲面。

依赖于抽象patch类型，图元生成器计算不同数量的细分曲面程度并应用不同的细分曲面算法。每个所生成的顶点在一个抽象patch内具有一个规格化的位置（即，坐标范围在[0, 1]之内）。这个位置具有两个或三个分量，依赖于patch的类型。这些坐标通过内建的

输入提供给细分曲面计算着色器。

**细分曲面计算着色器**

OpenGL细分曲面流水线的最后一个阶段就是细分曲面计算着色器执行。绑定的细分曲面计算着色器对图元生成器发射的每个细分曲面坐标逐个执行，并负责确定从细分曲面坐标所得到的顶点的位置。我们将看到，细分曲面计算着色器看上去与顶点着色器类似，将顶点变换到屏幕位置（除非细分曲面着色器的数据将被进一步由几何着色器来处理）。

配置一个细分曲面计算着色器的第一步是配置图元生成器，这通过使用一个layout指示符来完成。其参数指定了细分曲面域与后续所生成的图元的类型；实体图元的面部朝向（用于做面剔除）；以及在图元生成期间如何应用细分曲面程度。

**指定图元生成域**

我们现在将描述细分曲面计算着色器的**in**这个layout的参数。首先，我们先谈论指定细分曲面域。我们之前已经提及过，一共有三种类型的域来生成细分曲面坐标：

• quads——单位正方形中的一个矩形域；域坐标：带有范围在[0, 1]内的u, v值的一个个坐标对(*u*, *v*)。

• triangles——使用重心坐标的一个三角形域；域坐标：带有范围在[0, 1]内的a、b、c三个值的坐标(a, b, c)，这里a+b+c=1。

• isolines——跨单位正方形的一组线；域坐标：u值范围在[0, 1]，v值范围在[0, 1)范围的(*u*, *v*)坐标对。

**指定生成图元的面部朝向**

与OpenGL中任何填充的图元一样，所发射的顶点的次序决定了图元的脸部朝向。由于我们在这种情况下不直接发射顶点，而只是让图元生成器为我们来做，不过我们需要告诉图元生成器我们图元的右手旋转方向。在layout指示符中，指定cw为顺时针旋转，ccw为逆时针。

**指定细分曲面坐标的空间**

此外，我们可以控制外部细分曲面程度的浮点值如何用在确定周边的细分曲面坐标生成上。（内部细分曲面程度受这些选项影响。）

• equal_spacing——细分曲面程度被裁减到[1, *max*]范围内，然后取整到下一个最大整数值。

• fractional_even_spacing——值被裁减到[2, *max*]范围内，然后取整到下一个最大偶整数值n。边然后被划分为n-2条等长部分，以及两个剩余部分，每个在一端，剩余部分长度可能比其它长度要短。

• fractional_odd_spacing——值被裁减到[1, *max*-1]范围内，然后取整到下一个最大奇整数值n。边然后被划分为n-2条等长部分，以及两个剩余部分，每个在一端，剩余部分长度可能比其它长度要短。

**额外的细分曲面计算着色器layout选项**

最后，如果我们想输出点，而不是等值线或填充区域的话，我们可以应用point_mode选项。该选项将为每个由细分曲面计算着色器所处理的顶点渲染一单个点。

在layout指示符内的选项的次序不重要。下面例子描述的是一个生成三角形域的图元，使用相等空间，逆时针方向的三角形，但只渲染点，而不是互联的图元。

```
layout (triangles, equal_spacing, ccw, point_mode) in;
```

**指定一个顶点的位置**

从细分曲面控制着色器输出的顶点（即，在gl_out数组中的gl_Position的值）在计算着色器中的gl_in变量中可用。当它们与细分曲面坐标相结合时，可以用于生成输出顶点的位置。

细分曲面坐标在变量gl_TessCoord中提供给计算着色器。在下面例子中我们使用相等空间划分的四边形来渲染一个简单的patch。在这个例子中，细分曲面坐标用于对表面进行上色，然后此例子也描述了如何计算顶点坐标。我们这里要注意的是，gl_in中的gl_Position相当于原始的patch每个顶点的坐标，而gl_TessCoord则是经过细分曲面之后的新增细分曲面顶点，这些顶点的坐标值是被规格化在[0, 1]范围内的。我们通过以原patch的顶点坐标与当前处理的细分曲面顶点坐标做相应的插值计算来获得此细分曲面顶点最后输出的坐标值。我们在细分曲面计算着色器中可以访问所有gl_in数组的元素，即每次调用可以访问当前patch所有输入的顶点坐标。

```
#version 410 core

layout (quads, equal_spacing, ccw) in;

out vec4 color;

void main(void)
{
    float u = gl_TessCoord.x;
    float omu = 1 - u;    // omu为1减去'u'
    float v = gl_TessCoord.y;
    float omv = 1 - v;    // omv为1减去'v'

    color = gl_TessCoord; // color最后给到片段着色器时，值为(gl_TessCoord.x,
gl_TessCoord.y, 0.0, 1.0)

    gl_Position = omu * omv * gl_in[0].gl_Position +
                  u * omv * gl_in[1].gl_Position +
                  u * v * gl_in[2].gl_Position +
                  omu * v * gl_in[3].gl_Position;
}
```

**细分曲面计算着色器变量**

与细分曲面控制着色器类似，细分曲面计算着色器也有gl_in数组，它是一个结构体数组，每个元素如下定义：

```
in gl_PerVertex {
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
} gl_in[gl_PatchVerticesIn];
```

输出顶点的数据被存放在一个接口块中，如下定义：

```
out gl_PerVertex {
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
};
```

**参考资料：**

《OpenGL Programming Guide Eighth Edition -- The Official Guide tyo Learning OpenGL, Version 4.3》

https://www.opengl.org/wiki/Tessellation