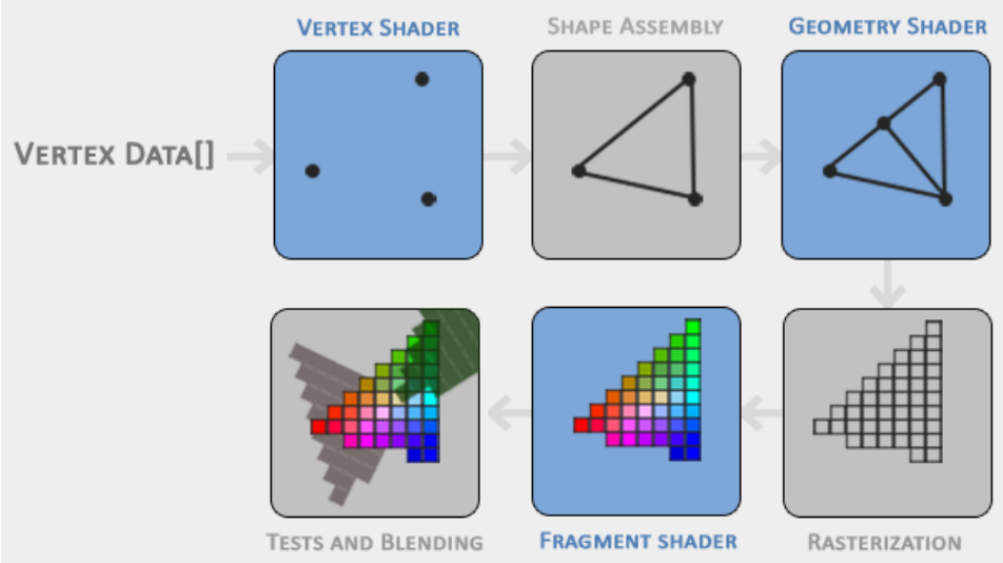
几何着色器

作者: 图形学搬运工
标签: 计算机图形学, 渲染管线



几何着色器概述 (Introduction)

几何着色器(Geometry Shader)是由第四代显卡着色器架构 Shader Model 4 正式引入的第三个着色器，属于渲染管线的一个可选阶段，位于曲面细分(Tessellation)和光栅化(Rasterization)之间。顶点着色器以顶点数据作为输入数据，而几何着色器则以完整的图元(Primitive)作为输入数据。例如，以三角形的三个顶点作为输入，然后输出对应的图元。与顶点着色器不能销毁或创建顶点不同，几何着色器的主要亮点就是可以创建或销毁几何图元，此功能让GPU可以实现一些有趣的效果。例如，根据输入图元类型扩展为一个或更多其他类型的图元，或者不输出任何图元。需要注意的是，几何着色器的输出图元不一定和输入图元相同。几何着色器的一个拿手好戏就是将一个点扩展为一个四边形(即两个三角形)。

几何着色器输出的图元由顶点列表定义而成，而且顶点必须变换到裁剪空间。也就是说，经过几何着色器处理后，得到的是一系列位于齐次裁剪空间的顶点所组成的图元。这些顶点会在后面的裁剪、透视除法和光栅化阶段得到进一步处理。

为了理解几何着色器是如何工作的，我们首先来看一个例子:

```
#version 150 core

layout(points) in;
layout(line_strip, max_vertices = 2) out;

void main()
{
    gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[0].gl_Position + vec4(0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    EndPrimitive();
}
```

- **输入类型 (Input Types)**

`layout(points) in;`

我们知道几何着色器的输入是图元，那么到底可以处理那些图元呢？可用的图元类型如下所示(括号里面的数字表示所需要的顶点数目)：

- points-GL_POINTS (1)
- lines-GL_LINES, GL_LINE_STRIP, GL_LINE_LIST （2)
- lines_adjacency-GL_LINES_ADJACENCY, GL_LINE_STRIP_ADJACENCY (4)
- triangles-GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN(3)
- triangles_adjacency-GL_TRIANGLES_ADJACENCY, GL_TRIANGLE_STRIP_ADJACENCY (6)

如果我们使用`glDrawArrays(GL_POINTS, 0, 4)`命令来绘制点图元，那么我们需要传入points参数。

- **输出类型 (Output Types)**

`layout(line_strip, max_vertices = 2) out;`

下一行代码展示的是几何着色器的输出。几何着色器最有意思的地方在于：输出图元类型跟输入图元类型完全不同，而且输出图元的数量跟输入图元数量也没有关系。括号里面的参数分别表示：输出图元类型和图元最大顶点数。输出图元的类型可以是下面几种：

- points
- line_strip
- triangle_strip

我们可以发现这三种类型的输出图元可以覆盖所有可能的图元类型，比如，我们可以用3个顶点的triangle_strip来输出一个普通的triangle。

几何着色器的输出要求我们最大输出顶点数目，当`EmitVertex()`的数量超出该值时，OpenGL将不会绘制更多的顶点。

- **顶点输入 (Vertex Input)**

我们在顶点着色器使用的`gl_Position`变量其实位于`gl_in`数组中，该数组的数据成员如下所示：

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

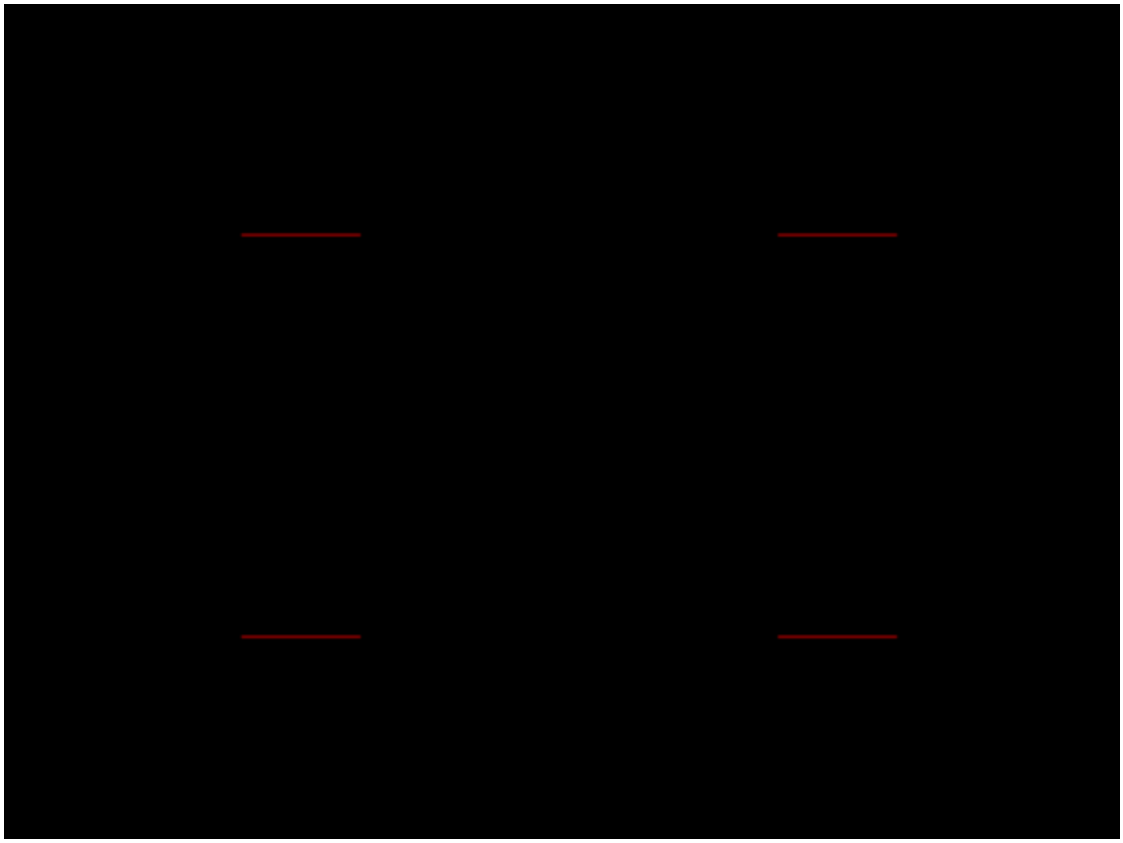
需要注意的是，由于几何着色器的输入是图元的所有顶点，所以这里将`gl_in`定义为数组。

- **顶点输出 (Vertex Output)**

我们使用`EmitVertex()`和`EndPrimitive()`两个函数来产生图元。每当我们调用一次`EmitVertex()`函数时，会将顶点加入到当前的图元；当所有顶点都加入到图元后，我们可以通过`EndPrimitive()`函数来产生图元。我们需要注意，当我们重复调用`EndPrimitive()`时，可以生成多个同样的图元。

```
void main()
{
    gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[0].gl_Position + vec4(0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    EndPrimitive();
}
```

通过执行该几何着色器，我们得到下面的输出，我们发现GPU的输出是一条线段，而不是顶点。



我们可以在顶点着色器中输入颜色分量来控制几何着色器中每个顶点的颜色。我们发现其实几何着色器对颜色的处理和片段着色器差不多，只不多这里的输入是颜色数组，因为输

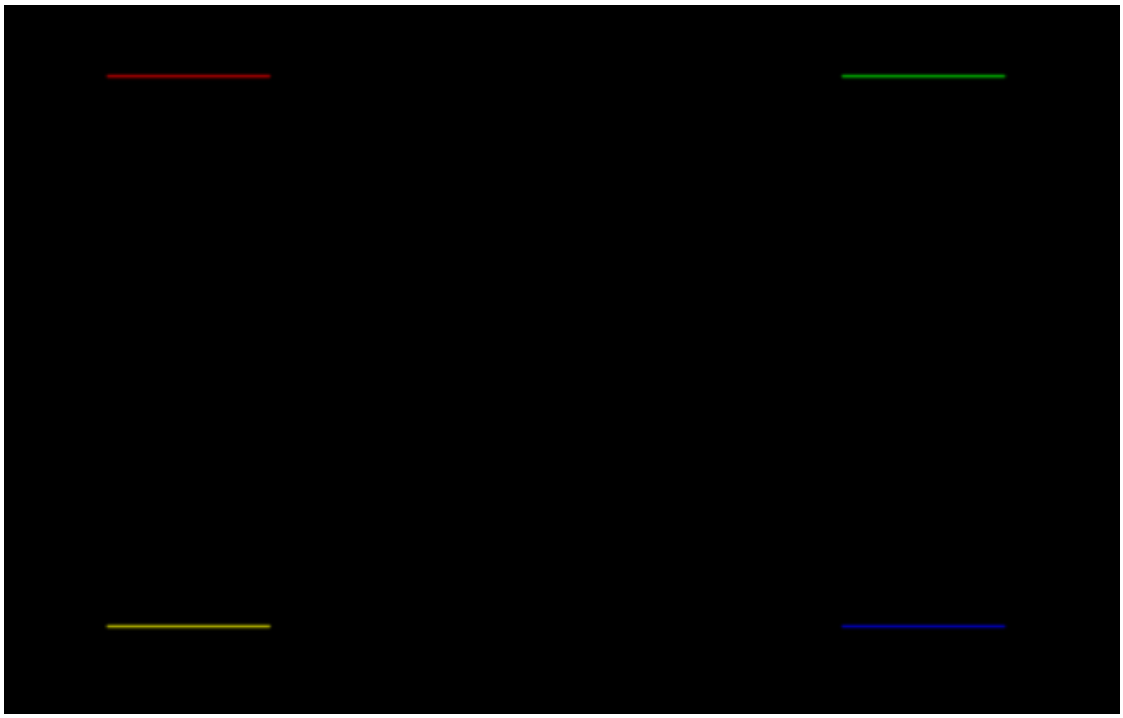
入图元的每个顶点都对应一个自己的颜色属性。

```
float points[] = {
    -0.45f,  0.45f, 1.0f, 0.0f, 0.0f, // Red point
     0.45f,  0.45f, 0.0f, 1.0f, 0.0f, // Green point
     0.45f, -0.45f, 0.0f, 0.0f, 1.0f, // Blue point
    -0.45f, -0.45f, 1.0f, 1.0f, 0.0f, // Yellow point
};
```

```
#version 150 core

in vec2 pos;
in vec3 color;

out vec3 vColor; // Output to geometry (or fragment) shader

void main()
{
    gl_Position = vec4(pos, 0.0, 1.0);
    vColor = color;
}
```

```
#version 150 core

layout(points) in;
layout(line_strip, max_vertices = 2) out;

in vec3 vColor[]; // Output from vertex shader for each vertex

out vec3 fColor; // Output to fragment shader

void main()
{
    ...
}
```

最终我们得到如下的结果，每个线段都有自己的颜色，而且与输入顶点的颜色相同。



几何着色器应用 (Application)

法线可视化

几何着色器的第一个应用是显示物体的法线，这对于光照效果的调试非常有帮助。我们首先在不使用几何着色器的情况下正常渲染一次场景；然后开启几何着色器第二次渲染场景，送到几何着色器的是三角形图元，我们为其每个顶点生成一个法线向量。

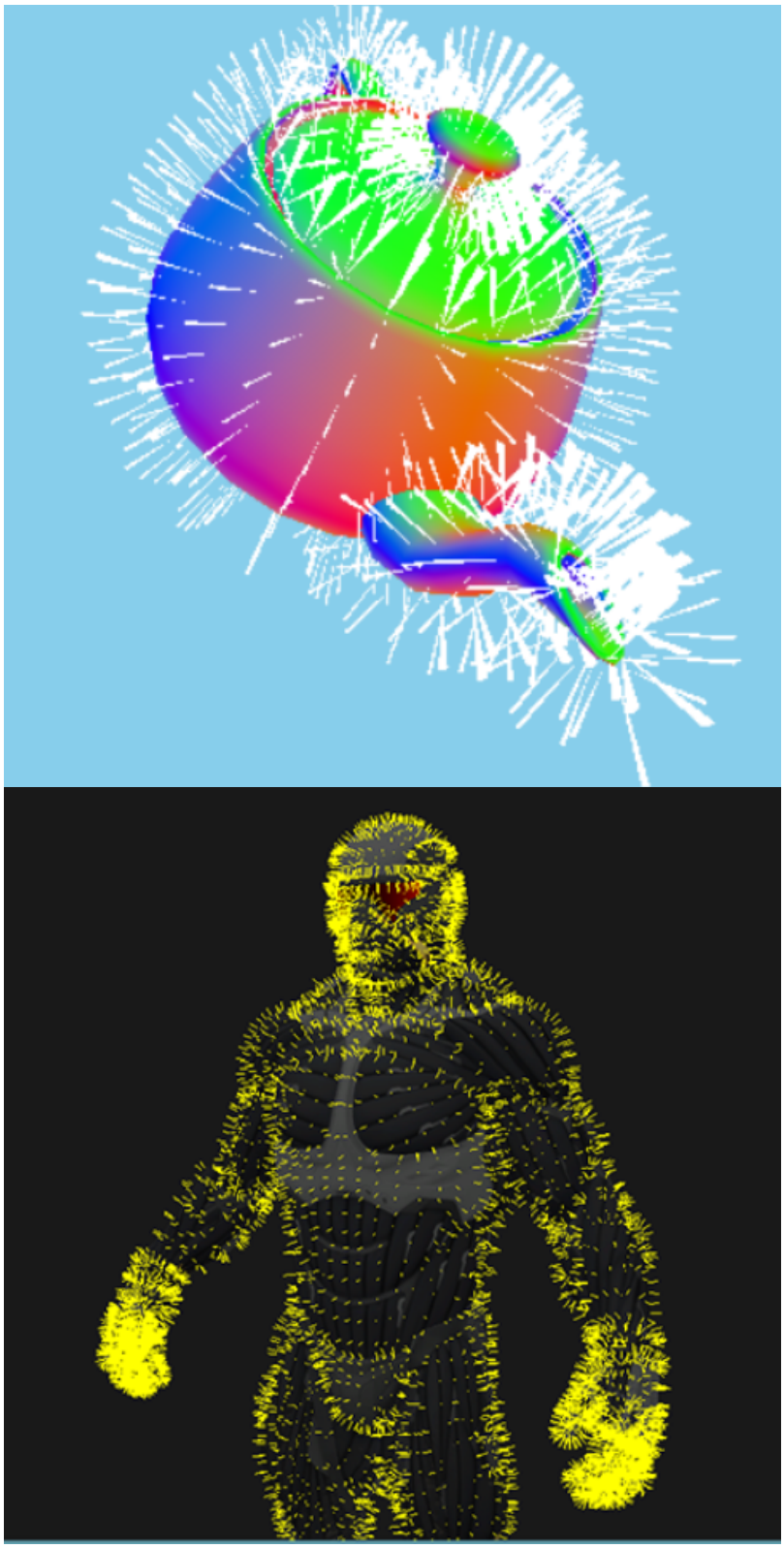
这里需要注意的是，几何着色器中使用的顶点坐标是经过顶点着色器变换后的裁剪空间坐标，所以传入到几何着色器的法线也需要变换到裁剪空间。我们这里需要知道法线向量的变换矩阵与顶点坐标的变换矩阵是不同的，需要使用矩阵的逆阵的转置来变换法向量，关于该矩阵的具体推导可以参考这个教程。顶点着色器如下所示，将法向量变换到裁剪空间。

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;

out VS_OUT {
    vec3 normal;
} vs_out;

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    mat3 normalMatrix = mat3(transpose(inverse(view * model)));
    vs_out.normal = normalize(vec3(projection * vec4(normalMatrix * aNormal, 0.0)));
}
```

在几何着色器中，我们接受的输入图元是triangle，输出的图元是line_strip，我们这里输出的顶点数量max_vertices为6，因为我们需要为triangle的每个顶点输出法线，三个法线向量与三角形的法线相互平行。从下面的几何着色器中我们看到，这里为triangle每个顶点输出了两个顶点，分别是法线的起点和终点，用来表示法线。

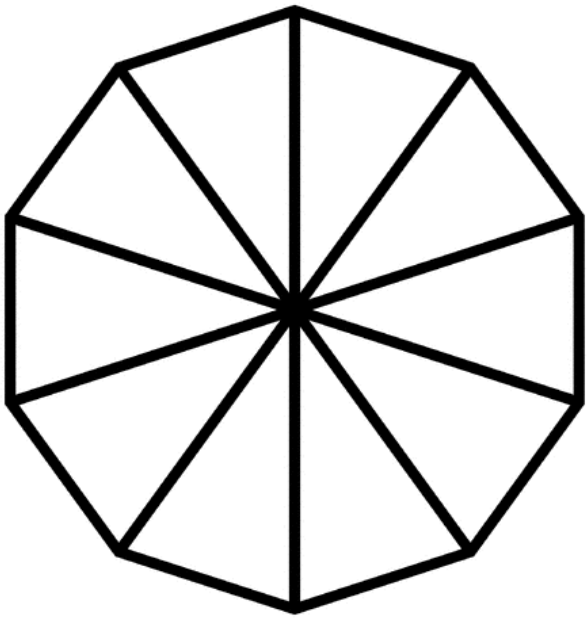
```
#version 330 core
layout (triangles) in;
layout (line_strip, max_vertices = 6) out;

in VS_OUT {
    vec3 normal;
} gs_in[];

const float MAGNITUDE = 0.4; //用来显示输出法线的大小

void GenerateLine(int index)
{
    gl_Position = gl_in[index].gl_Position;
    EmitVertex();
    gl_Position = gl_in[index].gl_Position + vec4(gs_in[index].normal, 0.0) * MAGNITUDE;
    EmitVertex();
    EndPrimitive();
}

void main()
{
    GenerateLine(0); // 第一个顶点法线
    GenerateLine(1); // 第二个顶点法线
    GenerateLine(2); // 第三个顶点法线
}
```

最终我们可以得到如下的效果图，我们通过几何着色器来可视化茶壶和机器人的法线，我们注意到，这样的效果类似于给物体增加了毛发(Fur)，所以几何着色器也经常用来实现毛发渲染。



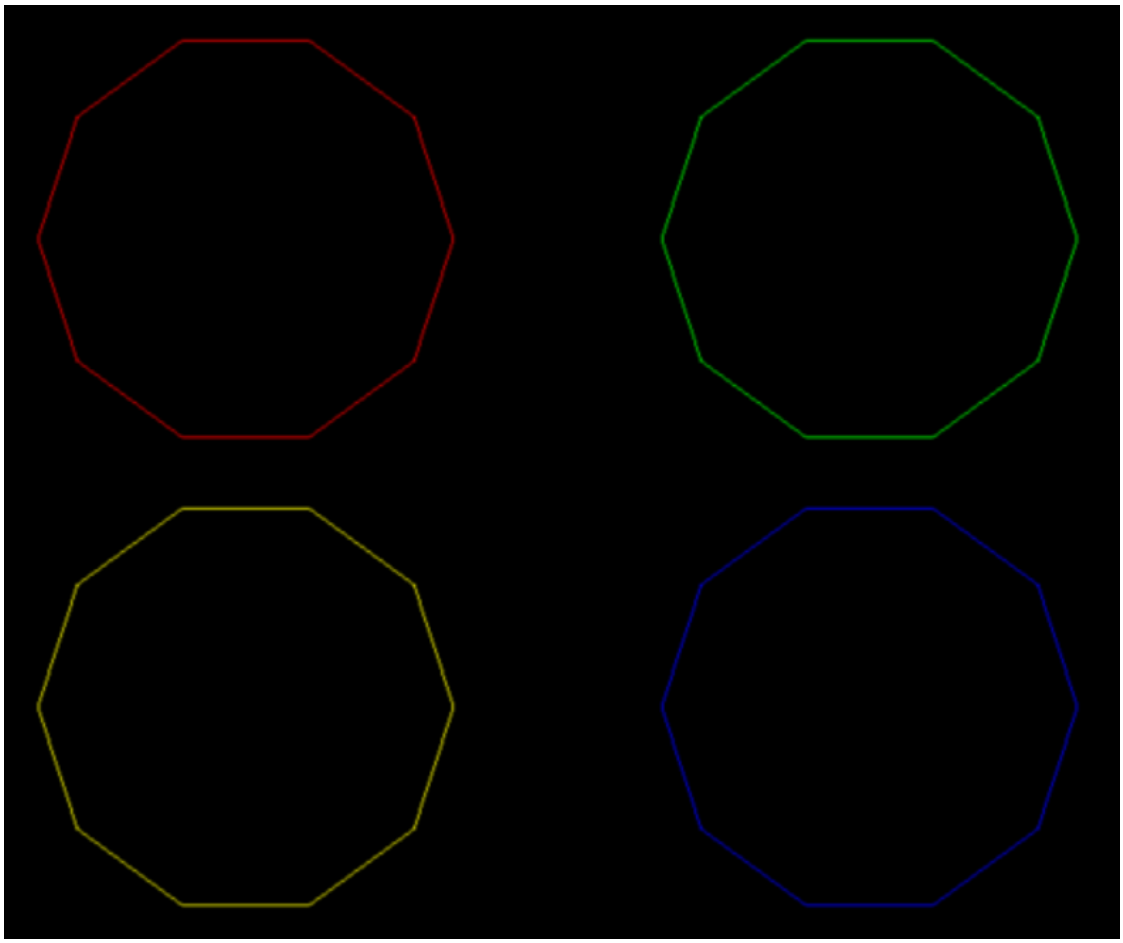
动态几何体形成 (Dynamically generating geometry)

接下来我们看下几何着色器的另一个应用：动态几何体形成。我们利用几何着色器可以实现物体的LOD技术 (Level of Detail)。比如，我们需要在游戏中绘制一个圆圈，那么我们可以根据距离摄像机的远近来调整圆圈的顶点数目，充分利用显卡的性能。我们首先来绘制一个十多边形 (10-sided polygon)，这里需要使用到三角学的一些知识，如下图所示。



```
#version 150 core

layout(points) in;
layout(line_strip, max_vertices = 11) out;

in vec3 vColor[];
out vec3 fColor;

const float PI = 3.1415926;

void main()
{
    fColor = vColor[0];

    for (int i = 0; i <= 10; i++) {
        // Angle between each side in radians
        float ang = PI * 2.0 / 10.0 * i;

        // Offset from center of point (0.3 to accomodate for aspect ratio)
        vec4 offset = vec4(cos(ang) * 0.3, -sin(ang) * 0.4, 0.0, 0.0);
        gl_Position = gl_in[0].gl_Position + offset;

        EmitVertex();
    }

    EndPrimitive();
}
```

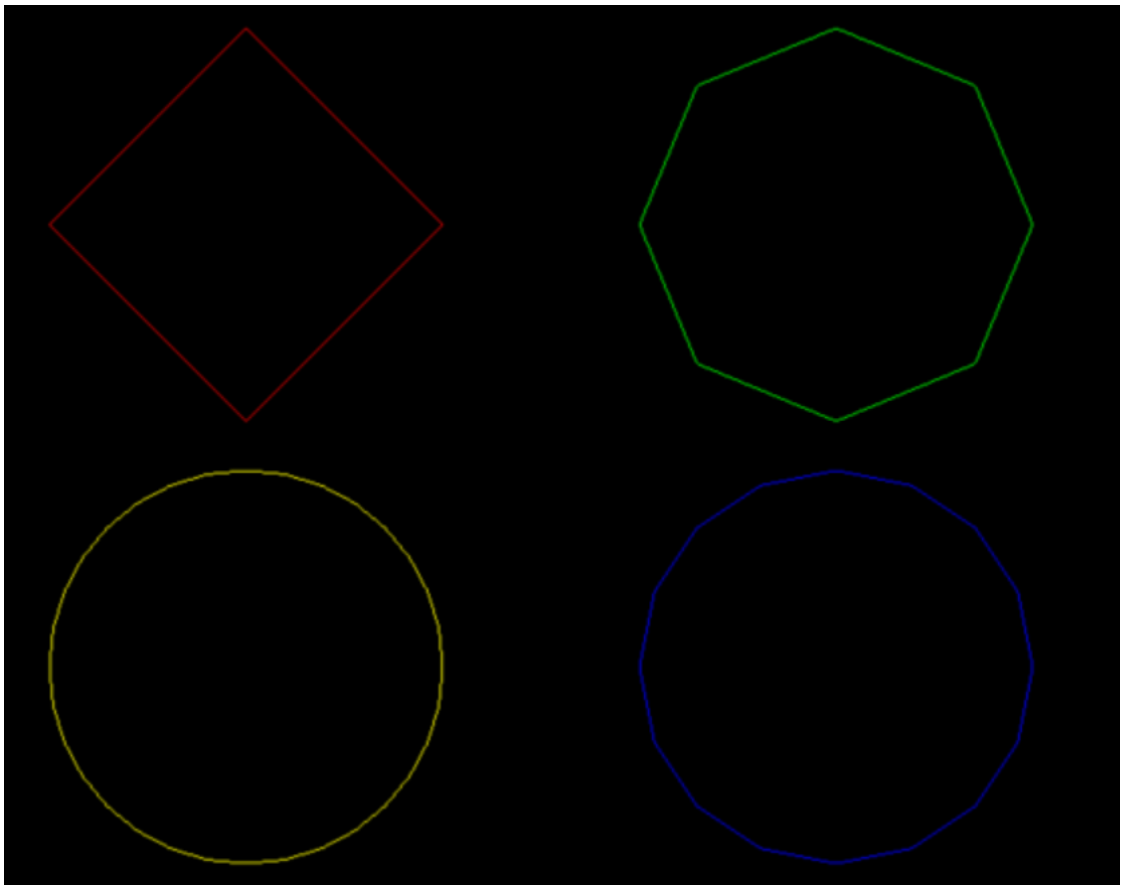
十多边形有十个顶点，这里我们绘制了十一个顶点，因为我们需要将多边形封闭起来，所以第一个顶点需要绘制两次。我们可到下面的结果：



接下来我们通过顶点数据来控制多边形的边数，也就是将边数变成顶点的一个属性值：

```
float points[] = {
//  Coordinates  Color             Sides
    -0.45f,  0.45f, 1.0f, 0.0f, 0.0f,  4.0f,
     0.45f,  0.45f, 0.0f, 1.0f, 0.0f,  8.0f,
     0.45f, -0.45f, 0.0f, 0.0f, 1.0f, 16.0f,
    -0.45f, -0.45f, 1.0f, 1.0f, 0.0f, 32.0f
};
```

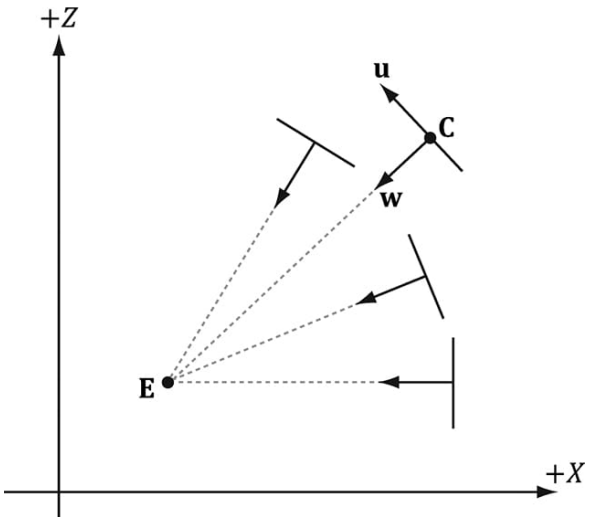
利用顶点的Sides数值来控制多边形的边数而不是原来的定值10，我们可以得到下面的效果图。四个多边形分别具有4、8、16和32条边。我们在游戏中可以根据玩家距离摄像机的远近来动态调整多边形边数这个属性值，实现LOD的效果。



公告牌 (BillBoards)

公告牌技术就是以3D树木图片的四边形来代替3D树的渲染的技术。我们可以使公告牌总是面向摄像机，这样从远处看过去，公告牌不容易露出破绽，如上图所示。

假设y轴指向正上方，平面xz表示地面，则树木的公告牌立于xz平面切与y轴平行面向摄像机。下图是从y轴俯视看到的公告牌的坐标系。我们给定公告牌的中心位置C (我们可以根据几何着色器将该顶点扩展成四边形来展示公告牌)，摄像机的位置E，通过叉乘我们可以得到公告牌局部坐标系和世界坐标系的对应关系：



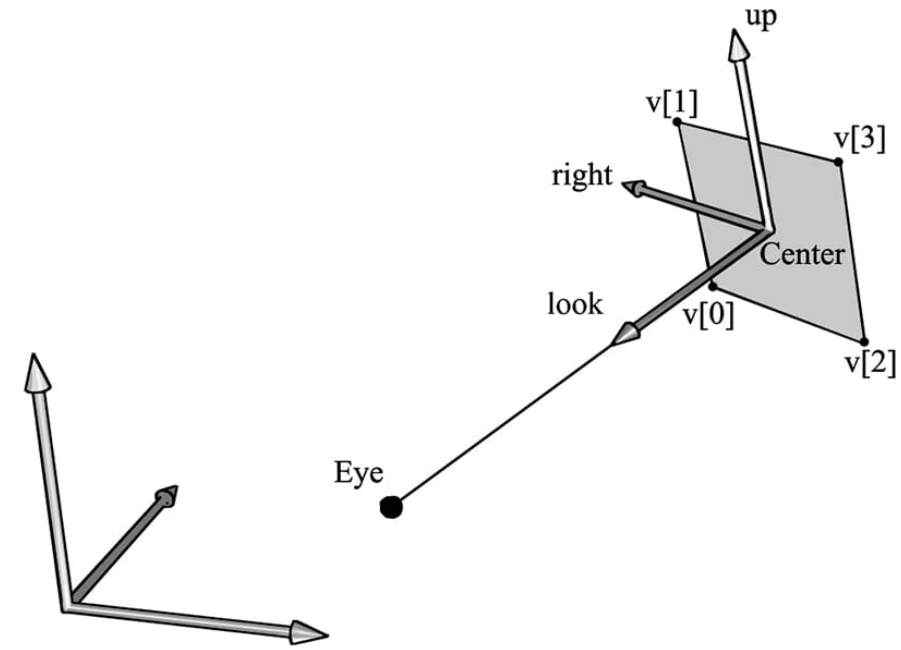
$$\mathbf{w} = \frac{\left(E_x - C_x, 0, E_z - C_z\right)}{\left\|\left(E_x - C_x, 0, E_z - C_z\right)\right\|}$$

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

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

通过公告牌的局部坐标系和世界坐标系之间的关系以及公告牌本身的大小，我们可以得到公告牌扩展后的四边形的四个顶点坐标如下：

```
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);
```



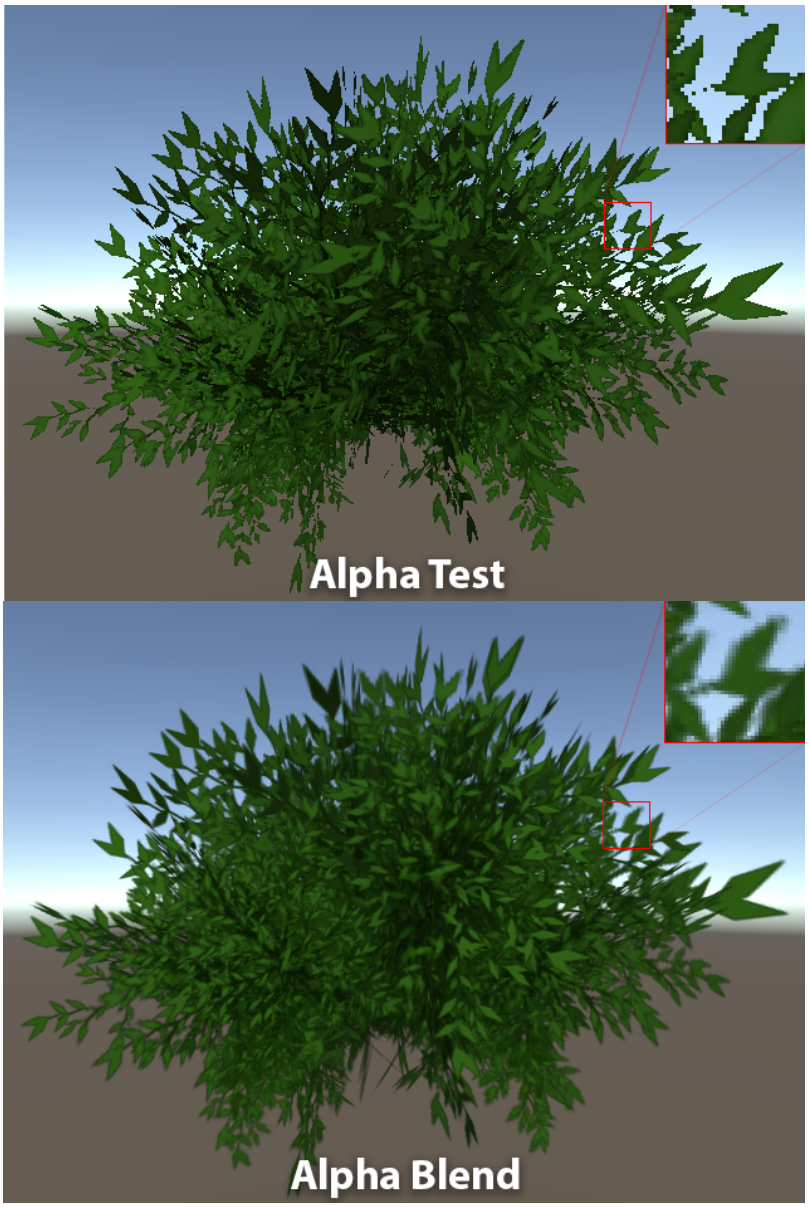
我们输入几何着色器的正是公告牌的中心点Center，通过几何着色器来扩展为四边形进行展示。

Alpha-to-Coverage技术

RGB Channel　　Alpha Channel

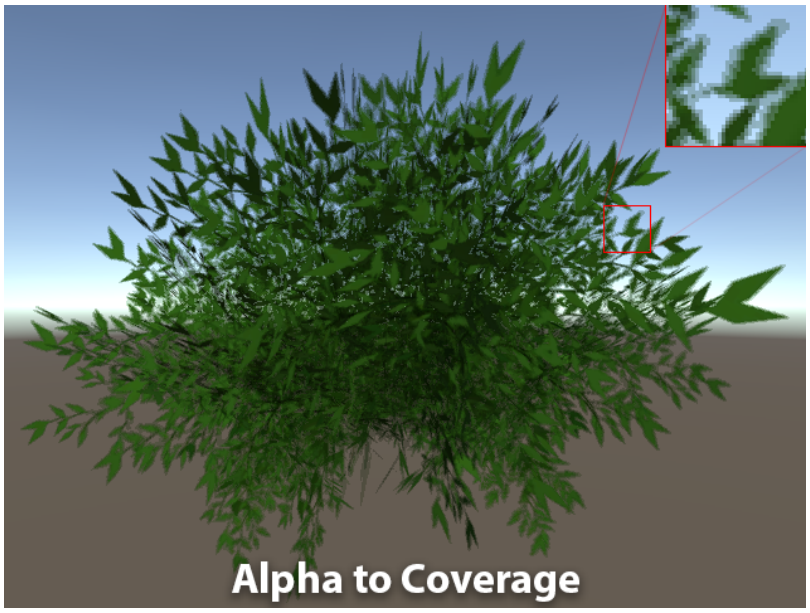
我们在BillBoards技术中会根据树木的alpha值进行alpha测试，当我们靠近观察公告牌时，会发现裁剪树木留下的硬边缘问题，也就是俗称的锯齿问题。这个问题主要是由于`discard()`或`clip()`函数带来的，它们用来裁剪掉不需要树木纹理的像素，导致树木的边缘过渡不自然。

解决该问题的第一种方法是使用alpha-blending技术来代替alpha test。通过线性纹理过滤，我们可以使得公告牌边缘过渡更加自然。由不透明像素逐渐过渡到半透明像素，带来一种渐变的效果。但是，运用alpha-blending技术需要将场景中的物体按照从后往前的顺序进行排序和渲染。对于渲染一大片的森林或草原，排序带来的消耗将非常大。而且alpha-blending技术会在排序后带来大量的OverDraw问题，会很大影响程序的性能。



第二种方法是使用多重采样抗锯齿技术 (Multisampling Antialiasing，MSAA)。可以用来缓解多边形边缘的锯齿问题，使之更加平滑。该技术确实有效，不过也带来一些问题。开启MSAA技术后，硬件会生成多个样本 (Samples)，根据样本的可见性和覆盖率来决定样本的颜色信息。关键在于，覆盖率是在多边形层次 (Polygon Level) 上确定下来的，所以MSAA技术并不会检查alpha通道，所以边缘过渡不会有不透明到半透明的过渡效果。如果想要在覆盖率计算过程中考虑alpha通道的情况，那就必须使用 Alpha-to-Coverage 的技术。Alpha-to-Coverage会使用纹理的alpha值来决定采样的覆盖率。

在开启了MSAA和alpha-to-coverage后，硬件会检测像素着色器返回的alpha值，并用于确定覆盖情况。比如本来利用MSAA得到的coverage是1，但是像素着色器返回的alpha为0.5，那么coverage变成了0.5，这样最后在resolve阶段，这个像素的颜色也被变淡了。正是通过这种技巧，将其颜色弱化，达到了柔软硬边的效果。关于alpha-to-coverage可以参考这篇文章。



当我们使用alpha测试来裁剪树叶或者围栏这类纹理时，建议使用MSAA和alpha-to-coverage技术来进行边缘过渡处理，所以我们发现其实alpha-to-coverage也属于抗锯齿技术的一种。

References

- Anti-aliased Alpha Test: The Esoteric Alpha To Coverage
- learnopengl
- Geometry Shaders