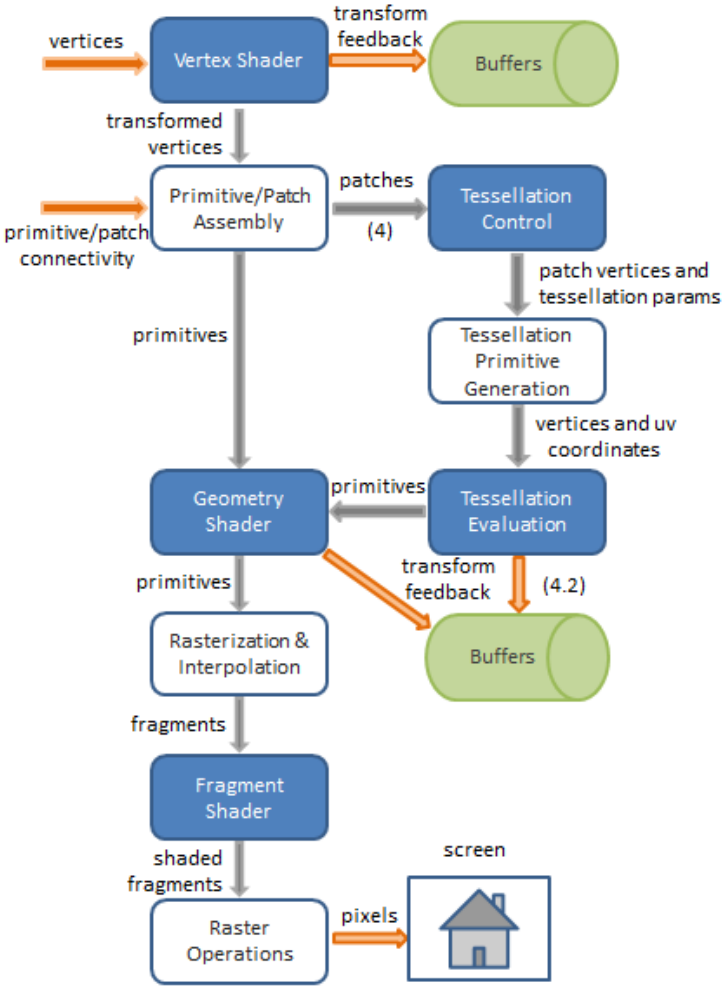
Universidad de Huelva

Tema 8

El shader de geometría

Índice

Índice


vertices
Vertex Shader
transform feedback
Buffers
transformed vertices
Primitive/Patch Assembly
patches
(4)
Tessellation Control
primitive/patch connectivity
patch vertices and tessellation params
primitives
Tessellation Primitive Generation
vertices and uv coordinates
Geometry Shader
primitives
Tessellation Evaluation
primitives
transform feedback
(4.2)
Rasterization & Interpolation
Buffers
fragments
Fragment Shader
shaded fragments
screen
Raster Operations
pixels

- Se trata de un shader opcional
- Se ejecuta una vez por cada primitiva generada en la etapa de ensamblado.
- El objetivo de este shader es realizar cambios sobre la primitiva ensamblada.
- Los cambios permiten sustituir la primitiva ensamblada por varias primitivas nuevas.
- Puede sustituirse por ninguna, es decir, eliminar la primitiva ensamblada.
- Las nuevas primitivas deben ser del mismo tipo.

- Ejemplos de uso del shader de geometría:
  - Eliminación de primitivas (por ejemplo, basada en técnicas de oclusión).
  - Incorporación de más nivel de detalle (por ejemplo, sustituyendo un triángulo por tres triángulos añadiendo un nuevo vértice basado en interpolaciones con vértices adyacentes)
  - Introducción de nuevas propiedades de los vértices, calculadas en base a información geométrica, para su uso posterior en el fragment shader.
  - Sustitución de la primitiva por otra totalmente diferente.

- El funcionamiento del GeometryShader se basa en dos funciones predefinidas:
  - *EmitVertex()* genera los vértices de las nuevas primitivas
  - *EndPrimitive()* indica que los vértices creados forman una primitiva y que los siguientes formarán una nueva.
- La función *EndPrimitive()* se llama implícitamente al terminar el programa del shader, de manera que se genera la primitiva con los últimos vértices emitidos.
- Si no se realiza ninguna llamada a *EmitVertex()*, el shader no genera ninguna primitiva y, por tanto, la primitiva original se elimina.

- El shader de geometría tiene como entrada la variable ***gl_in*[]**. Se trata de un array que almacena la información de cada vértice de la primitiva ensamblada.
- La estructura *gl_in* contiene los siguientes campos:
  - *gl_Position*: posición del vértice
  - *gl_PointSize*: tamaño de punto del vértice
  - *gl_ClipDistance*[]: distancias a planos clip (si se han activado)
- Como los valores de entrada son procesados por el shader, el valor de entrada de *gl_Position* puede estar descrito en cualquier sistema de coordenadas. Al generar los vértices de salida, el campo *gl_Position* debe describirse en coordenadas del clipping volume.

- Las variables de salida del VertexShader se definen como arrays de entrada en el GeometryShader. De esta forma en el shader de geometría se puede acceder a los valores de cualquier vértice de la primitiva. Por ejemplo:
  - VertexShader: **out vec3 VertexNormal;**
  - GeometryShader: **in vec3 VertexNormal[];**
- Las variables de salida del GeomertyShader se corresponden con las entradas interpoladas del FragmentShader. Al ejecutar la función *EmitVertex()*, el vértice generado utiliza los valores que estén almacenados en las variables de salida en ese momento.

- El tipo de primitiva que se asume como entrada se especifica mediante un modificador *layout* sobre la palabra clave *in*.

```
layout(XXXXXX) in;
```

- Se admiten los siguientes valores :
  - **points**: para primitivas GL_POINTS.
  - **lines**: para primitivas GL_LINES o GL_LINE_STRIP.
  - **lines_adjacency**: para primitivas GL_LINES_ADJACENCY o GL_LINE_STRIP_ADJACENCY.
  - **triangles**: para primitivas GL_TRIANGLES, GL_TRIANGLE_STRIP o GL_TRIANGLE_FAN.
  - **triangles_adjacency**: para primitivas GL_TRIANGLES_ADJACENCY o GL_TRIANGLE_STRIP_ADJACENCY.

- El tipo de primitiva a generar como salida se indica con un modificador *layout* aplicado a la palabra clave *out*.

```
layout(XXXXXX) out;
```

- Los valores admitidos son los siguientes:
  - **points**
  - **line_strip**
  - **triangle_strip**
- También es necesario indicar el número máximo de vértices a emitir en cada ejecución del shader añadiendo el valor de la propiedad *max_vertices* dentro del modificador *layout*.

- Se denomina "*point sprites*" (*puntos duende*) a dibujar con pequeñas figuras en lugar de puntos. Por ejemplo, sustituir los puntos por estrellitas, puntos gordos, caritas, etc.
- Una forma sencilla de producir las figuras es por medio de una textura. Si la figura no es cuadrada, los texels que no pertenecen a la figura serán transparentes.
- Vamos a ver un ejemplo de uso del GeometryShader para sustituir primitivas de puntos por cuadrados con textura.

- El VertexShader en este caso se limita a generar las coordenadas de los vértices en el sistema del observador. Para ello se utiliza como matriz de transformación la matriz ModelView.

```
#version 400

layout(location = 0) in vec3 VertexPosition;

uniform mat4 ModelView;

void main()
{
  gl_Position = ModelView * vec4( VertexPosition, 1.0);
}
```

- El GeometryShader recibe como entrada primitivas GL_POINTS y genera como salidas primitivas GL_TRIANGLE_STRIP con 4 vértices (lo que permite dibujar un cuadrado).

```
#version 400

layout(points) in;
layout(triangle_strip, max_vertices=4) out;

uniform mat4 Proj;
uniform float Size2;

out vec2 TexCoord;

void main()
{
   …
```

```
    vec4 desp0 = vec4(-Size2, -Size2, 0.0, 0.0);
    gl_Position = Proj * (desp0 + gl_in[0].gl_Position);
    TexCoord = vec2(0.0, 0.0);
    EmitVertex();

    vec4 desp1 = vec4(Size2, -Size2, 0.0, 0.0);
    gl_Position = Proj * (desp1 + gl_in[0].gl_Position);
    TexCoord = vec2(1.0, 0.0);
    EmitVertex();

    vec4 desp2 = vec4(-Size2, Size2, 0.0, 0.0);
    gl_Position = Proj * (desp2 + gl_in[0].gl_Position);
    TexCoord = vec2(0.0, 1.0);
    EmitVertex();

    vec4 desp3 = vec4(Size2, Size2, 0.0, 0.0);
    gl_Position = Proj * (desp3 + gl_in[0].gl_Position);
    TexCoord = vec2(1.0, 1.0);
    EmitVertex();

    EndPrimitive();
}
```

- El FragmentShader recibe como entrada las coordenadas de textura interpoladas para cada pixel y como variable uniforme la textura que queremos asociar al punto.

```
#version 400

in vec2 TexCoord;

uniform sampler2D SpriteTex;

out vec4 FragColor;

void main()
{
  FragColor = texture(SpriteTex, TexCoord);
}
```

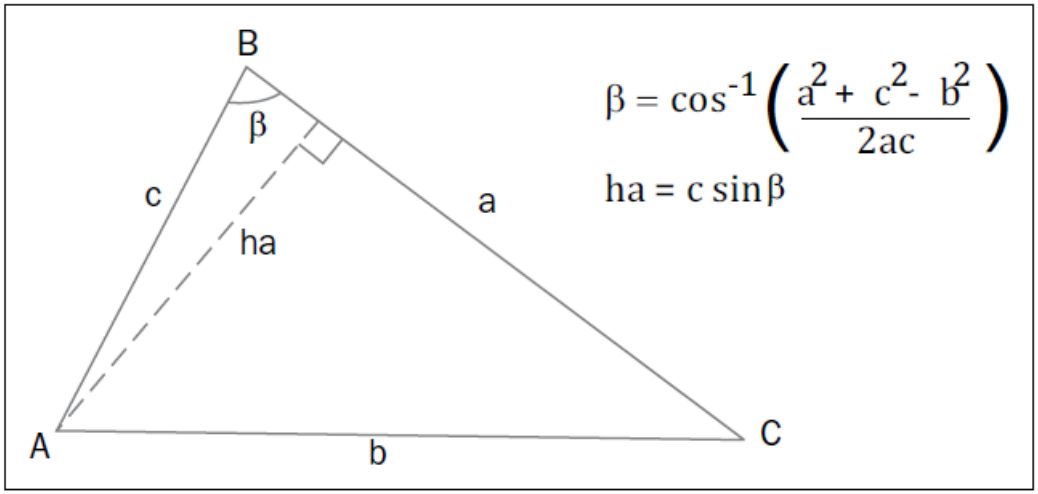
- Ejemplo de funcionamiento

(David Wolff - OpenGL 4 Shading Language Cookbook)

- El comando *glPolygonMode()* permite elegir el modo de dibujo entre GL_POINT, GL_LINE y GL_FILL.
- Si elegimos GL_LINE las primitivas triangulares se dibujarán en modo arista y los pixeles interiores no se dibujarán. Si elegimos GL_FILL las primitivas triangulares dibujarán todos los píxeles y no se distinguirán los bordes de los triángulos.
- Si queremos dibujar la arista de un color y el relleno de otro lo más sencillo es dibujar las primitivas dos veces, la primera en modo relleno y la segunda en modo arista. Para asegurarnos de que las aristas se dibujan encima del relleno se puede introducir un desplazamiento con *glPolygonOffset()*.

- A continuación vamos a desarrollar una forma diferente de dibujar las aristas sobre el relleno en cada triángulo por medio del shader de geometría.
- Con esta técnica no es necesario dibujar dos veces la primitiva, sino que el FragmentShader detecta cuando un pixel se encuentra sobre la arista o en el interior del triángulo y dibuja el pixel con el color adecuado.
- La técnica se basa en calcular la distancia de cada vértice al lado opuesto del triángulo y añadir esta distancia como atributo del vértice.
- Al interpolar los atributos, el FragmentShader puede saber la distancia del pixel a cada lado y decidir el color de dibujo.

- Para calcular la distancia (*ha*) de un vértice (*A*) al lado contrario (*a*) se utiliza la expresión:



- Esta expresión se obtiene considerando el teorema de Pitágoras sobre el triángulo rectángulo formado por *ha*, *b* y (*a-c·cos ß*).

```
#version 400

layout(triangles) in;
layout(triangle_strip, max_vertices=3) out;

noperspective out vec3 GEdgeDistance;
uniform mat4 Viewport;

void main()
{
   vec3 p0 = vec3(Viewport * (gl_in[0].gl_Position/
                              gl_in[0].gl_Position.w));
   vec3 p1 = vec3(Viewport * (gl_in[1].gl_Position/
                              gl_in[1].gl_Position.w));
   vec3 p2 = vec3(Viewport * (gl_in[2].gl_Position/
                              gl_in[2].gl_Position.w));
   float a = length(p1 - p2);
   float b = length(p2 - p0);
   float c = length(p1 - p0);
   …
```

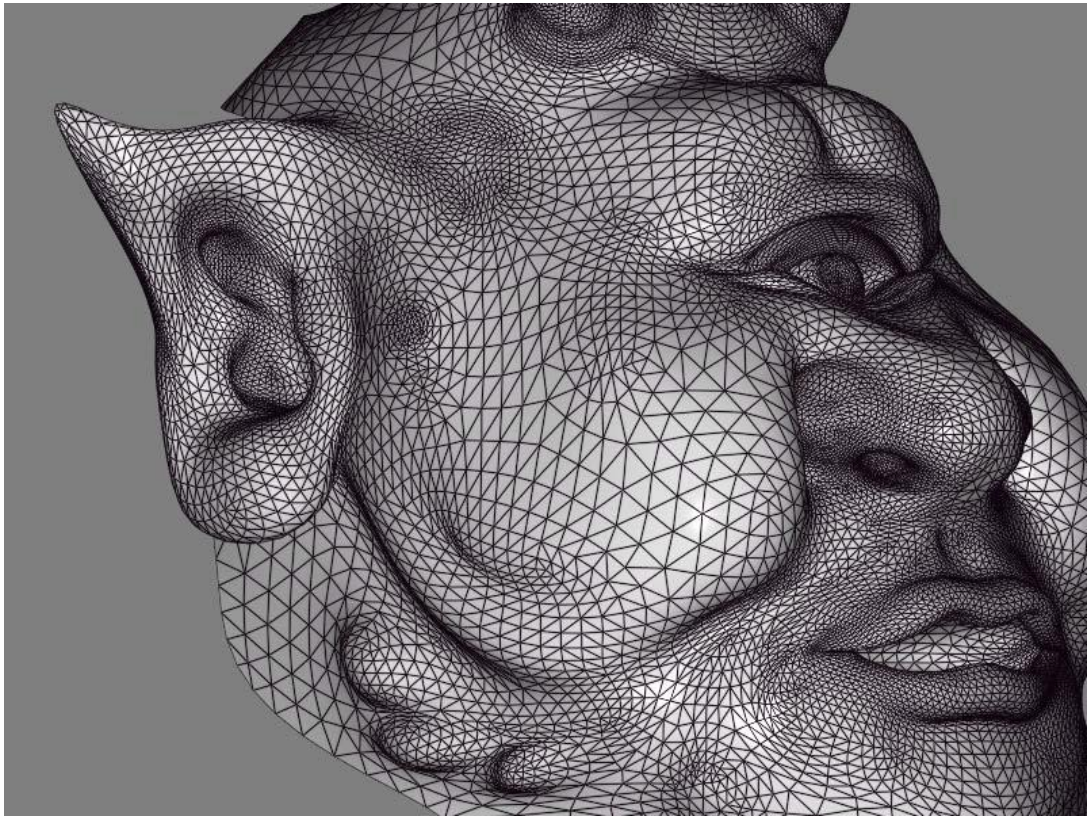
```
    …
    float alpha = acos( (b*b+c*c-a*a)/(2.0*b*c) );
    float beta = acos( (a*a+c*c-b*b)/(2.0*a*c) );
    float ha = abs( c * sin(beta) );
    float hb = abs( c * sin(alpha) );
    float hc = abs( b * sin(alpha) );

    GEdgeDistance = vec3( ha, 0.0, 0.0);
    gl_Position = gl_in[0].gl_Position;
    EmitVertex();

    GEdgeDistance = vec3( 0.0, hb, 0.0);
    gl_Position = gl_in[1].gl_Position;
    EmitVertex();

    GEdgeDistance = vec3(0.0, 0.0, hc);
    gl_Position = gl_in[2].gl_Position;
    EmitVertex();

    EndPrimitive();
}
```

```
#version 400

uniform float LineWidth;
uniform vec4 FillColor;
uniform vec4 LineColor;

noperspective in vec3 GEdgeDistance;

out vec4 FragColor;

void main()
{
  float d = min(GEdgeDistance.x, GEdgeDistance.y);
  d = min( d, GEdgeDistance.z);
  float mixVal = smoothstep(LineWidth - 1,
                            LineWidth + 1,d);
  FragColor = mix(LineColor, FillColor, mixVal);
}
```

- Sistema de referencia Viewport:
  - El proceso de rasterización se realiza considerando el sistema de referencia del clipping volume (coordenadas entre -1 y 1). La transformación final debe hacer corresponder cada punto del clipping volume a un pixel de la imagen. Es decir, hay que realizar una transformación al tamaño de la ventana (0,0,width,height).
- Modificador **noperspective**:
  - Por defecto, los valores de entrada del FragmentShader se obtienen por interpolación lineal entre los atributos de los vértices. Los coeficientes de interpolación se calculan a partir de la posición de cada vértice. Por defecto, los valores de la posición utilizados para calcular la interpolación son (p.x/p.w, p.y/p.w, p.z/p.w). La opción *noperspective* hace que los coeficientes de interpolación se calculen a partir de las posiciones (p.x,p.y,p.z).

- Función *smoothstep(a,b,c)*
  - Devuelve la relación de *c* respecto al intervalo (*a,b*). Es decir, si *c<a* devuelve 0; si *c>b* decuelve 1; si *a<c<b* devuelve *(c-a)/(b-a)*.
- Función *mix(a,b,c)*
  - Devuelve una mezcla de *a* y *b* con proporción *c*. Es decir, *(1-c)*a+c*b*.
- Es fácil realizar una modificación de estos shaders para incorporar los efectos de luz o texturas, en vez de utilizar un color de relleno fijo (*FillColor*).

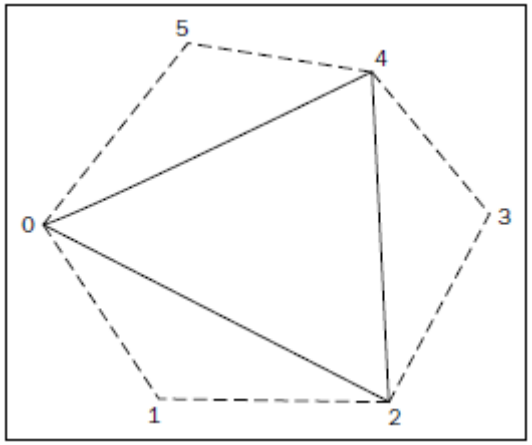
- Ejemplo de funcionamiento



(David Wolff - OpenGL 4 Shading Language Cookbook)

- El GeometryShader puede recibir como entrada no solo las propiedades de los vértices que forman cada primitiva, sino también las propiedades de vértices adyacentes a la primitiva.
- Para ello, además de puntos, líneas y triángulos, los comandos de dibujo pueden lanzar las siguientes primitivas:
  - GL_LINES_ADJACENCY: cada primitiva tiene acceso a 4 vértices, los dos que forman la línea y el vértice anterior y posterior.
  - GL_LINE_STRIP_ADJACENCY: una serie de n segmentos de línea se define en términos de n+3 vértices.
  - GL_TRIANGLES_ADJACENCY: cada triángulo se define con 6 vértices.
  - GL_TRIANGLE_STRIP_ADJACENCY: para n triángulos se requieren $2n$+4 vértices.
- Esto supone básicamente añadir índices a la lista de índices.

- El objetivo de este ejemplo es dibujar la silueta de los objetos. Esto supone detectar las aristas del objeto que representan los límites del objeto en la imagen.
- Para detectar las aristas de la silueta vamos a considerar un GeometryShader al que se le pasan primitivas GL_TRIANGLES_ADJACENCY.

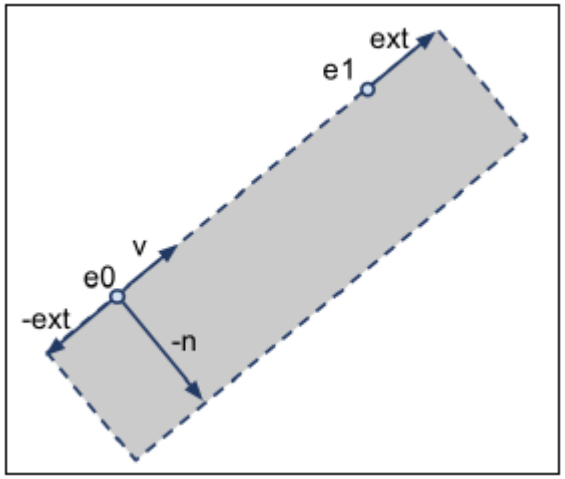


- Las aristas de las siluetas son aquellas en las que uno de los triángulos que comparten la arista se observa en su cara frontal y el otro triángulo se observa en su cara posterior.
- Para saber que cara se observa de cada triángulo hay que calcular el vector normal. Si la componente *z* del vector normal es positiva, entonces el triángulo se observa en su cara frontal. Si la componente *z* es negativa estamos viendo el triángulo en su cara posterior.
- Dado un triángulo ABC, la componente *z* del vector normal es

$$n_z = (A_x B_y - B_x A_y) + (B_x C_y - C_x B_y) + (C_x A_y - A_x C_y)$$

- El GeometryShader puede modificar primitivas o crear nuevas primitivas del mismo tipo, pero no puede crear nuevas primitivas de diferente tipo.
- Podríamos definir un shader que detectara las aristas de la silueta y emitiera primitivas GL_LINES. En tal caso solo se generarían las líneas de la silueta, pero no se dibujaría el interior de los objetos.
- Para dibujar tanto el interior de los objetos como las líneas de silueta es necesario que la salida sea primitivas triangulares.
- En el shader propuesto, las aristas de la silueta se van a dibujar por medio de rectángulos colocados sobre la arista. De esta forma la salida del shader serán primitivas triangulares.

- Para una arista situada entre los puntos ***e0*** y ***e1*** se va a generar un rectángulo (formado por dos triángulos) que envuelve a la arista con una prolongación de tamaño ***ext*** y un grosor de tamaño ***n***. La extensión se va a calcular como un porcentaje de la longitud de la arista (***PCTExtend***) y el grosor se indica con la variable ***EdgeWidth***.



```
#version 400

layout(triangles_adjacency) in;
layout(triangle_strip, max_vertices=15) out;

flat out bool GIsEdge;

uniform float EdgeWidth;
uniform float PctExtend;

bool isFrontFacing(vec3 a, vec3 b, vec3 c)
{
    nz = (a.x*b.y - b.x*a.y)+(b.x*c.y - c.x*b.y)
         + (c.x*a.y - a.x*c.y);
    return (nz >0);
}

…
```

```
…

void emitEdgeQuad(vec3 e0, vec3 e1)
{
  vec2 ext = PctExtent * (e1.xy - e0.xy);
  vec2 v = normalize(e1.xy - e0.xy);
  vec2 n = vec2(-v.y,v.x)*EdgeWidth;
  GIsEdge = true;
  gl_Position = vec4(e0.xy-ext,e0.z,1.0);
  EmitVertex();
  gl_Position = vec4(e0.xy-n-ext, e0.z, 1.0);
  EmitVertex();
  gl_Position = vec4(e1.xy + ext, e1.z, 1.0);
  EmitVertex();
  gl_Position = vec4(e1.xy-n+ext, e1.z, 1.0);
  EmitVertex();

  EndPrimitive();
}

…
```

```
…

void main()
{
  vec3 p0 = gl_in[0].gl_Position.xyz /
            gl_in[0].gl_Position.w;
  vec3 p1 = gl_in[1].gl_Position.xyz /
            gl_in[1].gl_Position.w;
  vec3 p2 = gl_in[2].gl_Position.xyz /
            gl_in[2].gl_Position.w;
  vec3 p3 = gl_in[3].gl_Position.xyz /
            gl_in[3].gl_Position.w;
  vec3 p4 = gl_in[4].gl_Position.xyz /
            gl_in[4].gl_Position.w;
  vec3 p5 = gl_in[5].gl_Position.xyz /
            gl_in[5].gl_Position.w;

  …
```

```
  …

  if(isFrontFacing(p0, p2, p4) {
    if( ! isFrontFacing(p0,p1,p2)) emitEdgeQuad(p0,p2);
    if( ! isFrontFacing(p2,p3,p4)) emitEdgeQuad(p2,p4);
    if( ! isFrontFacing(p4,p5,p0)) emitEdgeQuad(p4,p0);
  }

  GIsEdge = false;
  gl_Position = gl_in[0].gl_Position;
  EmitVertex();
  gl_Position = gl_in[2].gl_Position;
  EmitVertex();
  gl_Position = gl_in[4].gl_Position;
  EmitVertex();

  EndPrimitive();
}
```

```
#version 400

uniform vec4 FillColor;
uniform vec4 LineColor;

flat in bool GIsEdge;

out vec4 FragColor;

void main()
{
  if(GIsEdge) {
   FragColor = LineColor;
  } else {
   FragColor = FillColor;
  }
}
```

- Ejemplo de funcionamiento

(David Wolff - OpenGL 4 Shading Language Cookbook)