Programando la GPU (IV)

Shaders de teselación

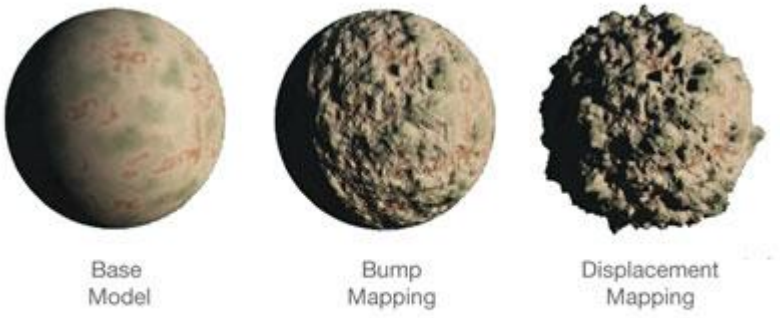
flickr.com/photos/guypaterson

Bibliografía:

- Superbiblia, 7ª ed. pp. 33-37 y 305-333

Motivación

- La teselación permite aumentar la geometría de un modelo en tiempo de ejecución dentro de la GPU
  - Para dar más detalle geométrico donde realmente hace falta, sin aumentar el tráfico en el bus
- *Displacement mapping* (transformación de los vértices generados proceduralmente, o a partir de una textura)



nvidia.com/object/tessellation.html

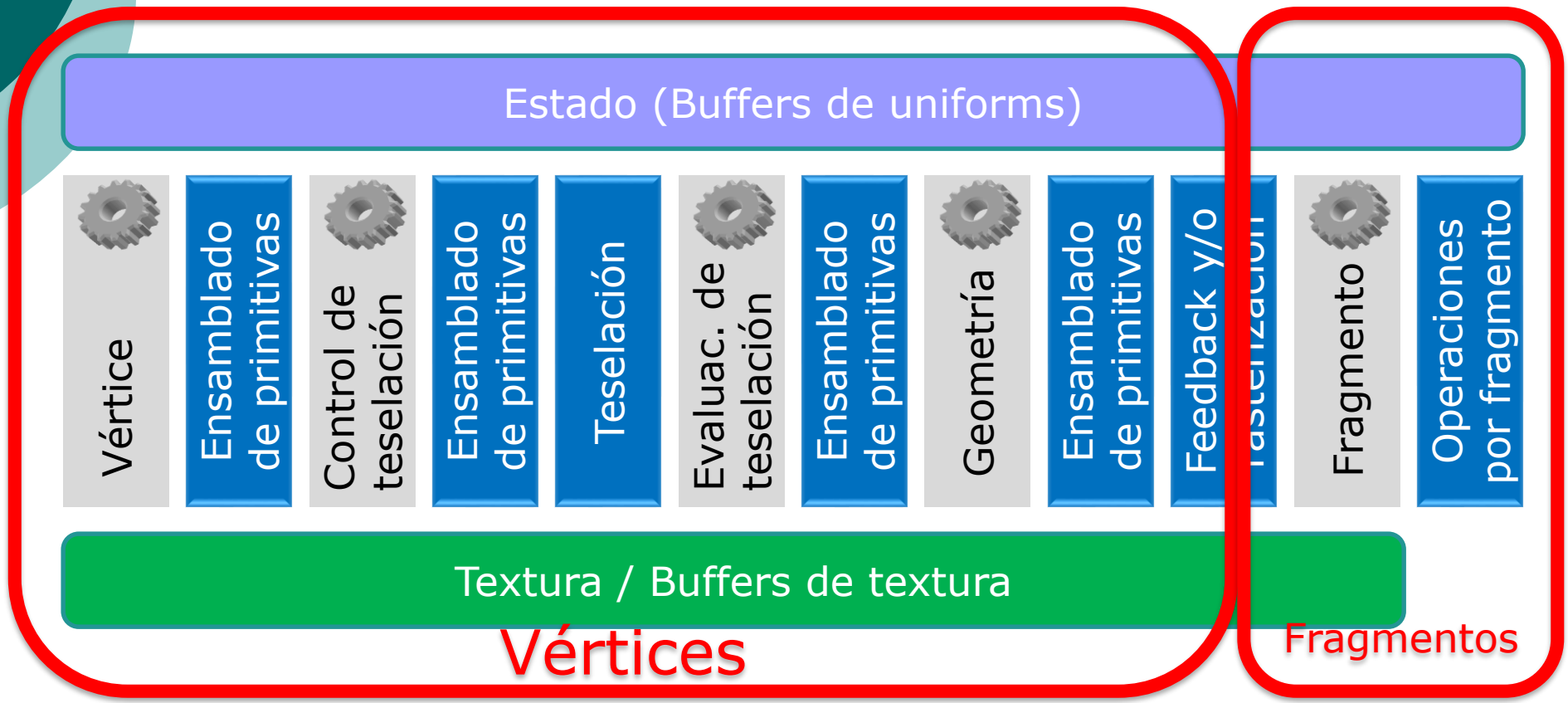
Teselación en OpenGL

- Los shaders de teselación no trabajan con las primitivas básicas de OpenGL, sino con parches
  - Es un error dibujar un parche sin shaders de teselación, o dibujar algo que no sea un parche (p.e., triángulo), con un shader de teselación
- La teselación dividirá el parche de entrada en puntos, líneas o triángulos
- El shader de vértice recibe los vértices del parche, y genera la salida para el shader de control de teselación

Teselación en OpenGL

- La teselación tiene tres etapas (con dos tipos de shaders distintos)
  - Control: su entrada son los vértices de los parches, y decide cuánta geometría generar. Son opcionales
  - Etapa fija con el motor de teselación
  - Evaluación: calcula las coordenadas de la malla generada (y el resto de sus atributos)

Tubería de OpenGL 4.x



Teselación en OpenGL

- Un parche es una lista ordenada de vértices o puntos de control (es el shader el que decide su significado)
- Se usan las mismas funciones de dibujo que para el resto de primitivas (`glDrawArrays()`, `glDrawElements()`…)
  - Se usa el tipo de primitiva `GL_PATCHES`
  - En la llamada se establece el total de vértices a procesar
  - Se usa `glPatchParameteri(GL_PATCH_VERTICES, n)` para indicar cuántos vértices tiene cada parche (el mismo número para todos los parches de una llamada). Por defecto n=3, y el máximo es, al menos, 32

Teselación en OpenGL

- Ambos tipos de shaders procesan un vértice cada vez (no como los shaders de geometría que procesan todos los vértices en cada ejecución)

Shaders de control de teselación

- Recibe un parche emitido por la aplicación (y procesado por el shader de vértice), y genera un parche para la siguiente etapa de teselación
- Es opcional
- Funciones:
  1. Preparar los vértices (puntos de control) para el shader de evaluación de teselación
  2. Quizá actualizar atributos de vértice o de parche
  3. Especificar los factores de nivel de teselación que controla el generador de primitivas. Definen cuánto teselar el parche. Se pueden definir en tiempo de ejecución

Shaders de control de teselación

1. Preparar los vértices para el shader de evaluación de teselación
   - Entrada: `gl_in[]`, salida: `gl_out[]`
   - Puede modificar, añadir y eliminar vértices de los parches
   - Especifican el número de vértices de salida con:
     - `layout (vertices = 16) out;`
   - El número de vértices de salida indica el número de veces que se ejecutará el shader
   - `gl_InvocationID` identifica qué vértice de salida se está procesando (se debería escribir en dicha posición de `gl_out`)

Shaders de control de teselación

- Variables predefinidas:
  - `in int gl_InvocationID`
  - `in int gl_PrimitiveID`
  - `in int gl_PatchVerticesIn`
  - `patch out float gl_TessLevelOuter[4];`
  - `patch out float gl_TessLevelInner[2]`

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

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

Shaders de control de teselación

1. Preparar los vértices para el shader de evaluación de teselación
   - Se tiene acceso a todos los vértices, tanto de entrada como de salida (aunque sólo se debe escribir en la posición del array que le toca)
     - Problema de sincronización: la función `barrier()` obliga a que todas las ejecuciones del TCS de un parche se sincronicen en ese punto

Shaders de control de teselación

1. Ejemplo del TCS identidad:

```
#version 420 core

layout (vertices = 4) out;

void main() {
  gl_out[gl_InvocationID].glPosition =
        gl_in[gl_InvocationID].gl_Position;
  // Falta establecer los niveles de teselación
}
```

Shaders de control de teselación

- Además, se pueden definir arrays de atributos adicionales:
  - De entrada: un atributo `out` del shader de vértice se convierte en un array (de tamaño `gl_PatchVerticesIn`, o sin tamaño)
  - De salida (de tamaño `gl_PatchVerticesOut`, o sin tamaño)
- También se pueden definir atributos por parche, usando el calificador `patch`

Shaders de control de teselación

3. Especificar los factores de nivel de teselación
   - Hay tres dominios de teselación (que definen cómo se teselarán los patches):
     - Cuadriláteros, triángulos e isolíneas
   - El nivel de teselación se controla mediante dos variables:
     - `float gl_TessLevelOuter[4]`: define cuánto se subdivide el perímetro del parche
     - `float gl_TessLevelInner[2]`: cuánto subdividir el interior del parche
     - El número de elementos utilizados de los arrays anteriores depende del tipo de teselación

Shaders de control de teselación

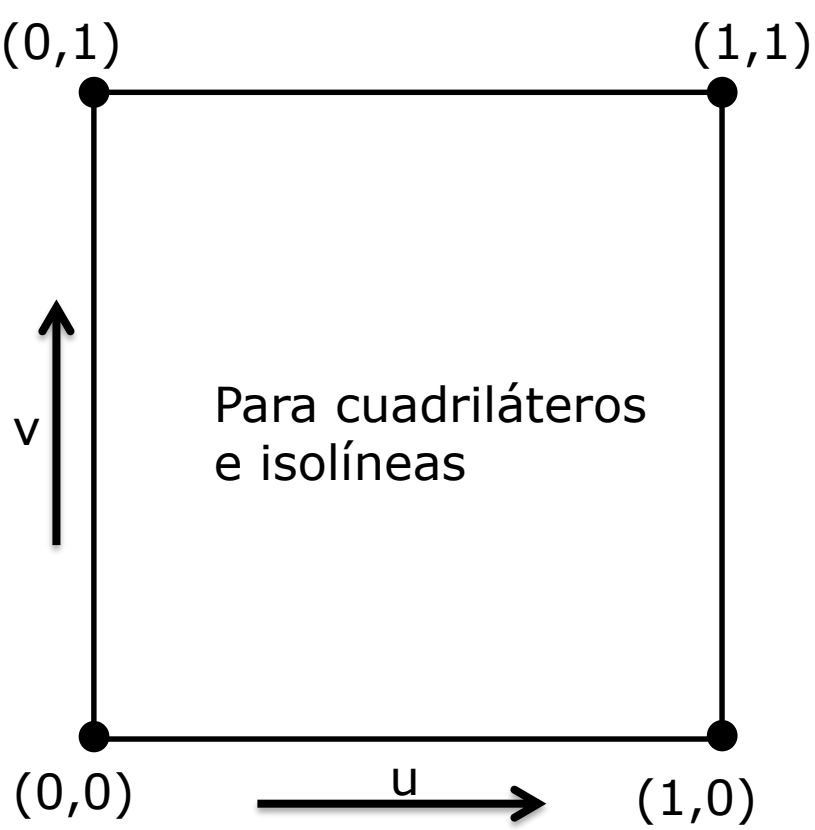
- Si no hay un TCS instalado, se usan los niveles de teselación por defecto, establecidos con:
- `void glPatchParameterfv(pname, float *values)`
  - `pname: GL_PATCH_DEFAULT_OUTER_LEVEL` o `GL_PATCH_DEFAULT_INNER_LEVEL`
  - `values`: un array de 4 o 2 floats, respectivamente
  - Por defecto, `{1, 1, 1, 1}` y `{1, 1}`
  - Un nivel de teselación de 0 descarta el parche

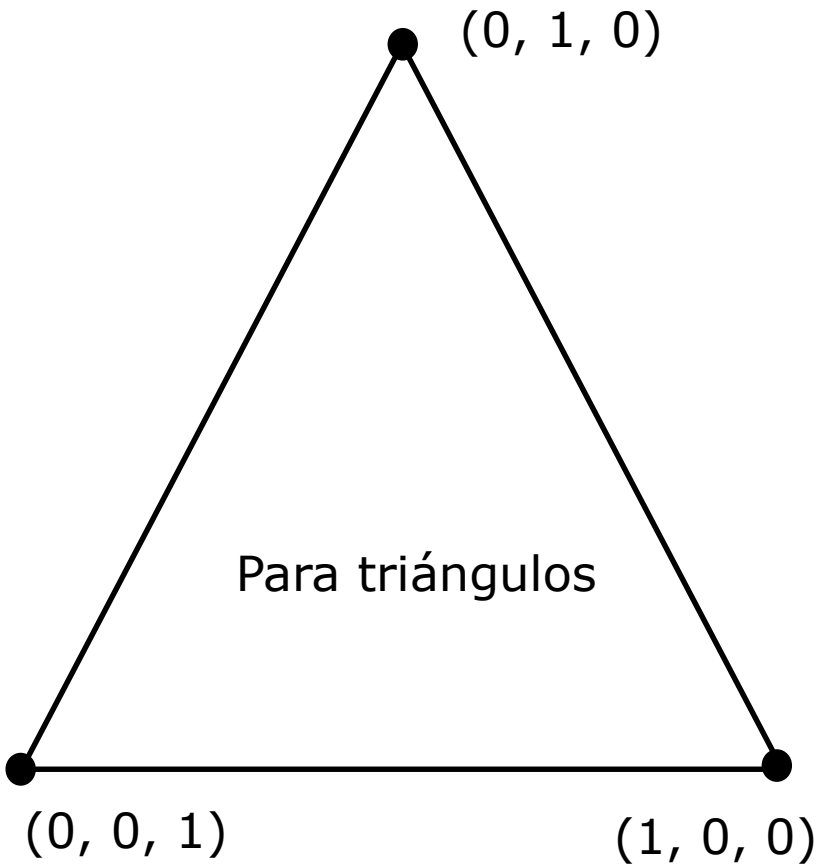
Coordenadas de teselación

- La función fija de teselación subdivide el parche de entrada en triángulos, líneas o puntos
- Cada vértice generado en el proceso de teselación lleva asignada una coordenada de teselación, en un espacio parámetrico (*uv* para rectángulos e isolíneas, y *uvw* para triángulos)
- Todas las coordenadas de teselación están entre 0 y 1

Coordenadas de teselación

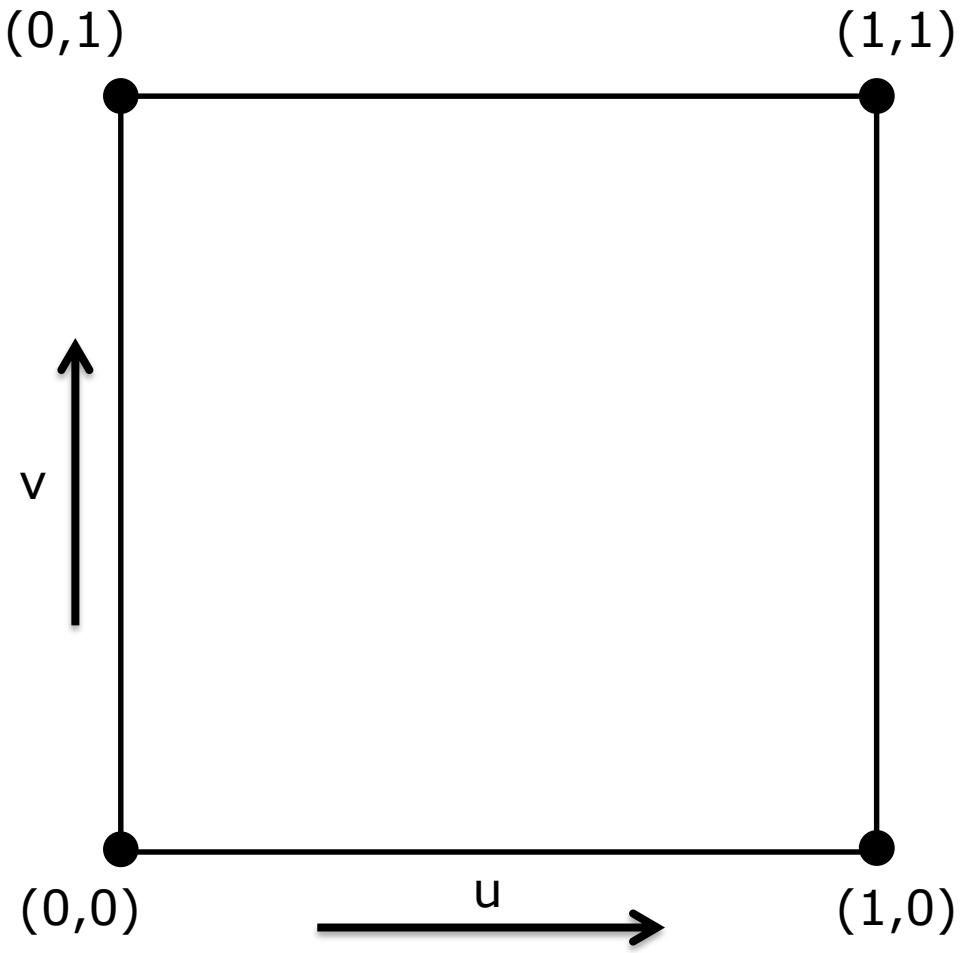
(u, v) y (u, v, w)





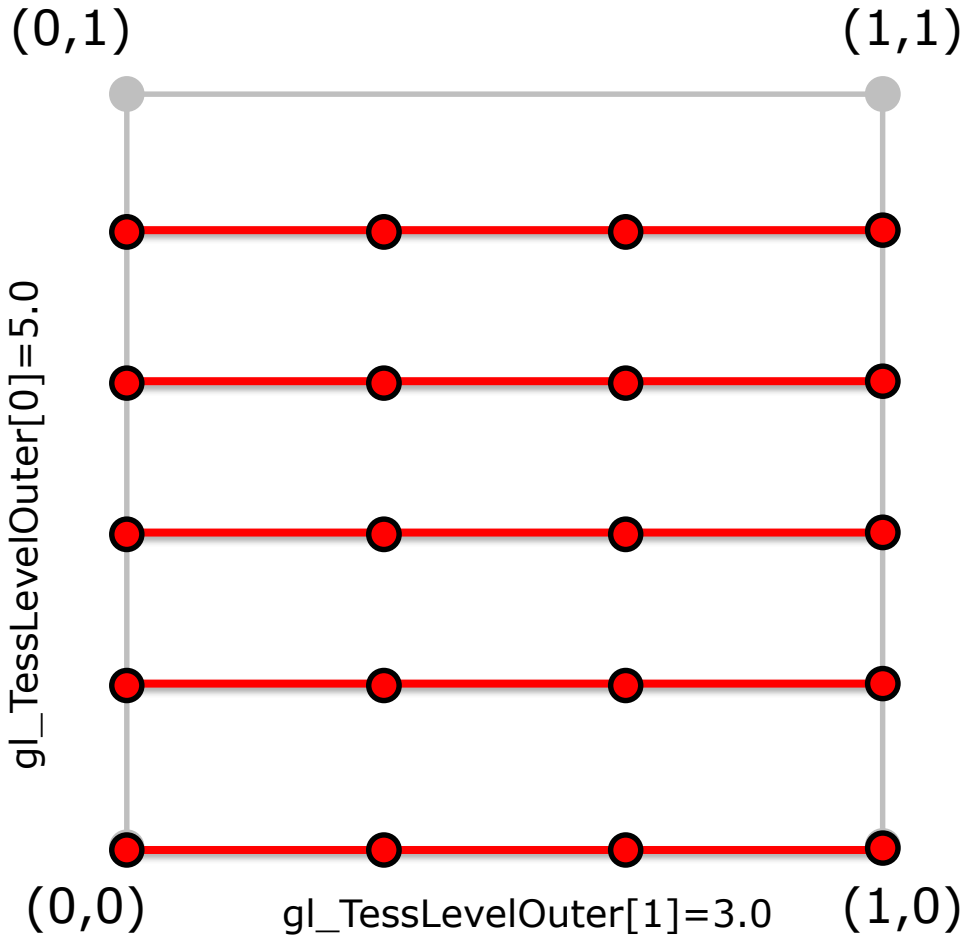
Teselación de isolíneas

Espacio paramétrico (u,v)



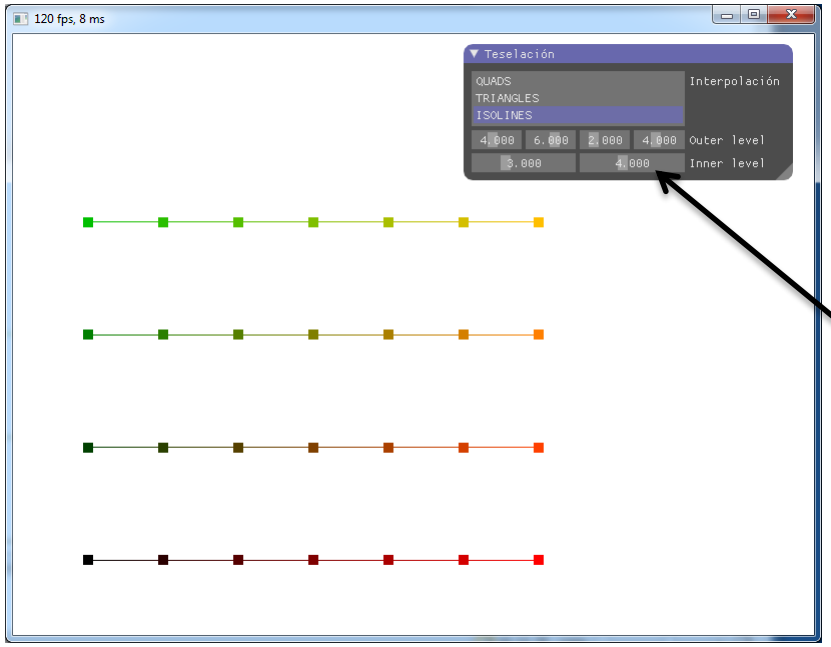
Teselación de isolíneas

Espacio paramétrico (u,v)



Teselación de isolíneas

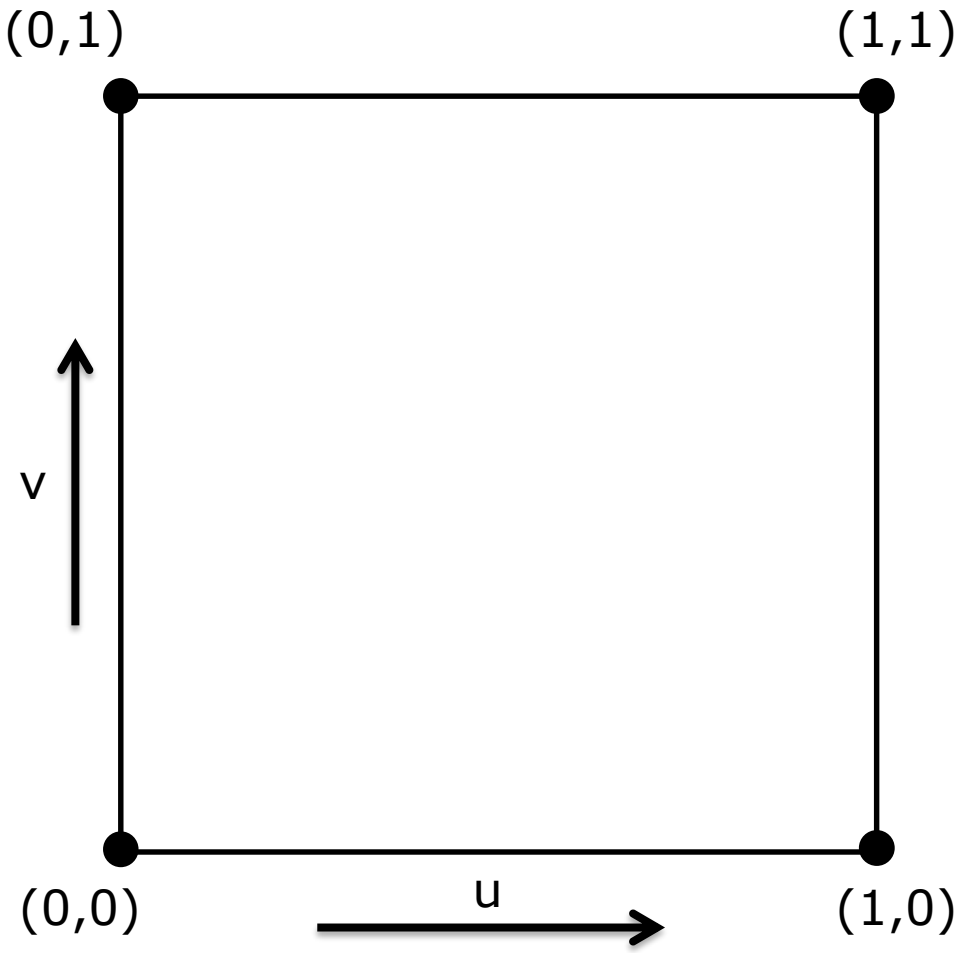
ej7-2



Puedes hacer Ctrl+Clic para introducir un valor por teclado

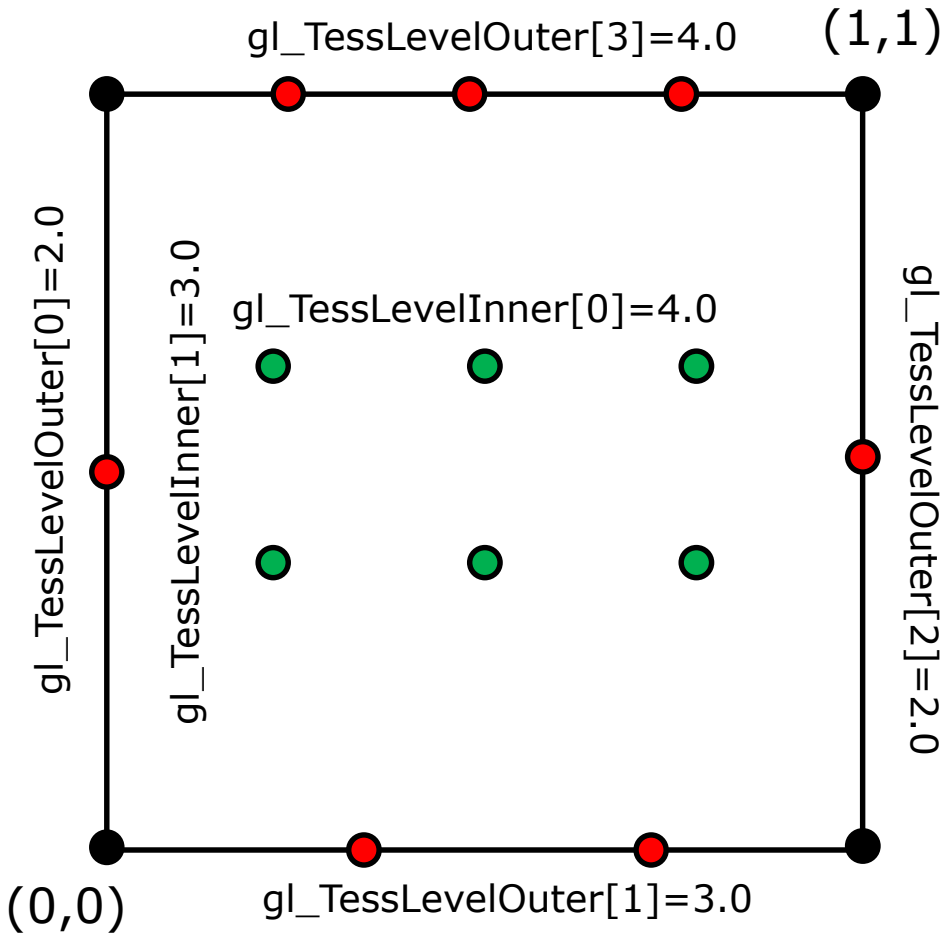
Teselación de cuadriláteros

Espacio paramétrico (u,v)



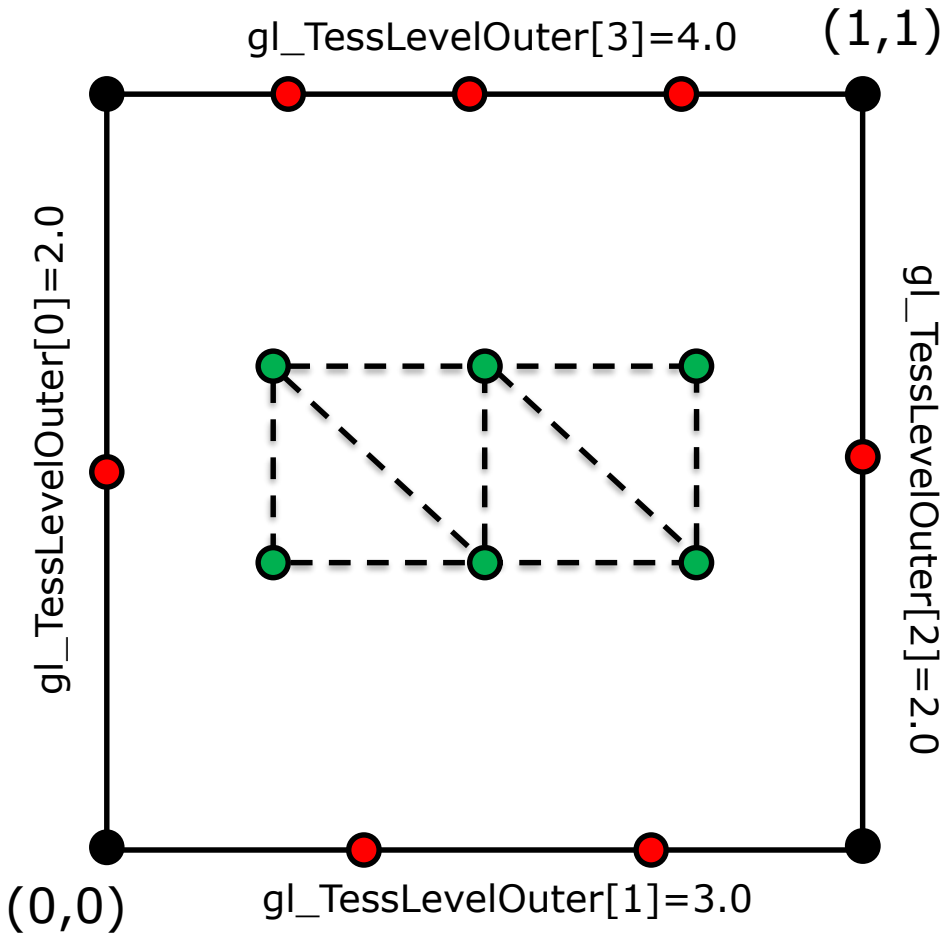
Teselación de cuadriláteros

Espacio paramétrico (u,v)



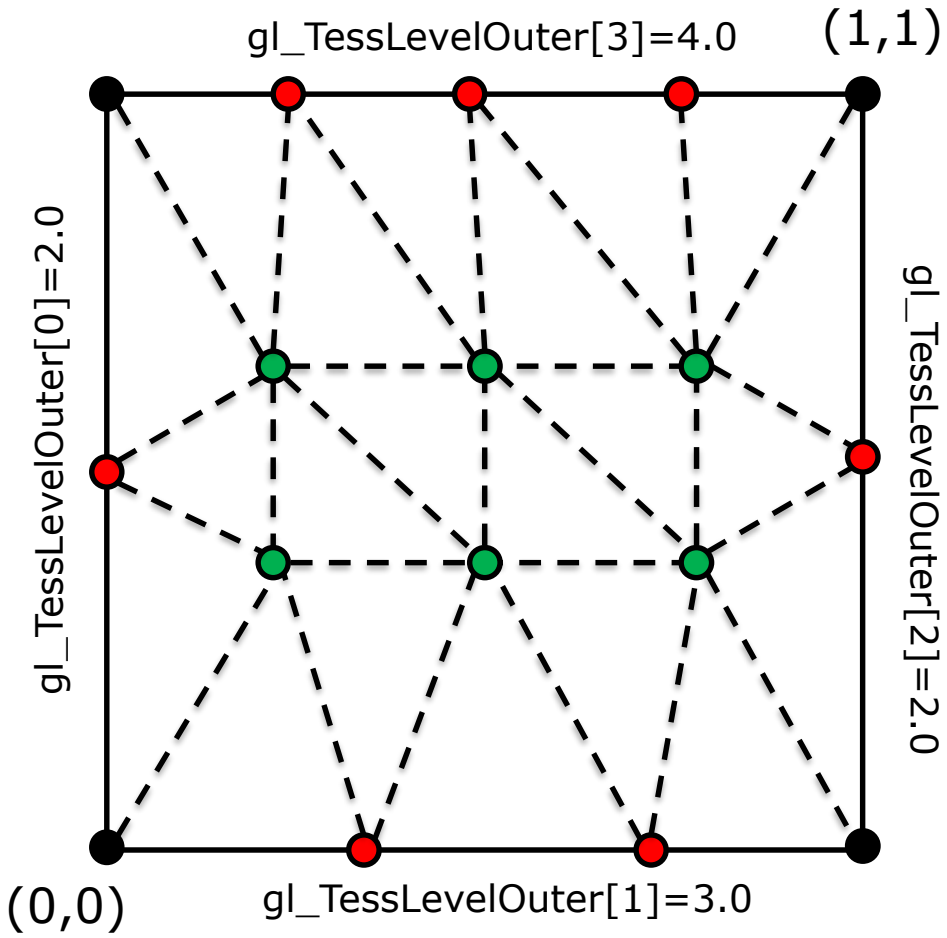
Teselación de cuadriláteros

Espacio paramétrico (u,v)



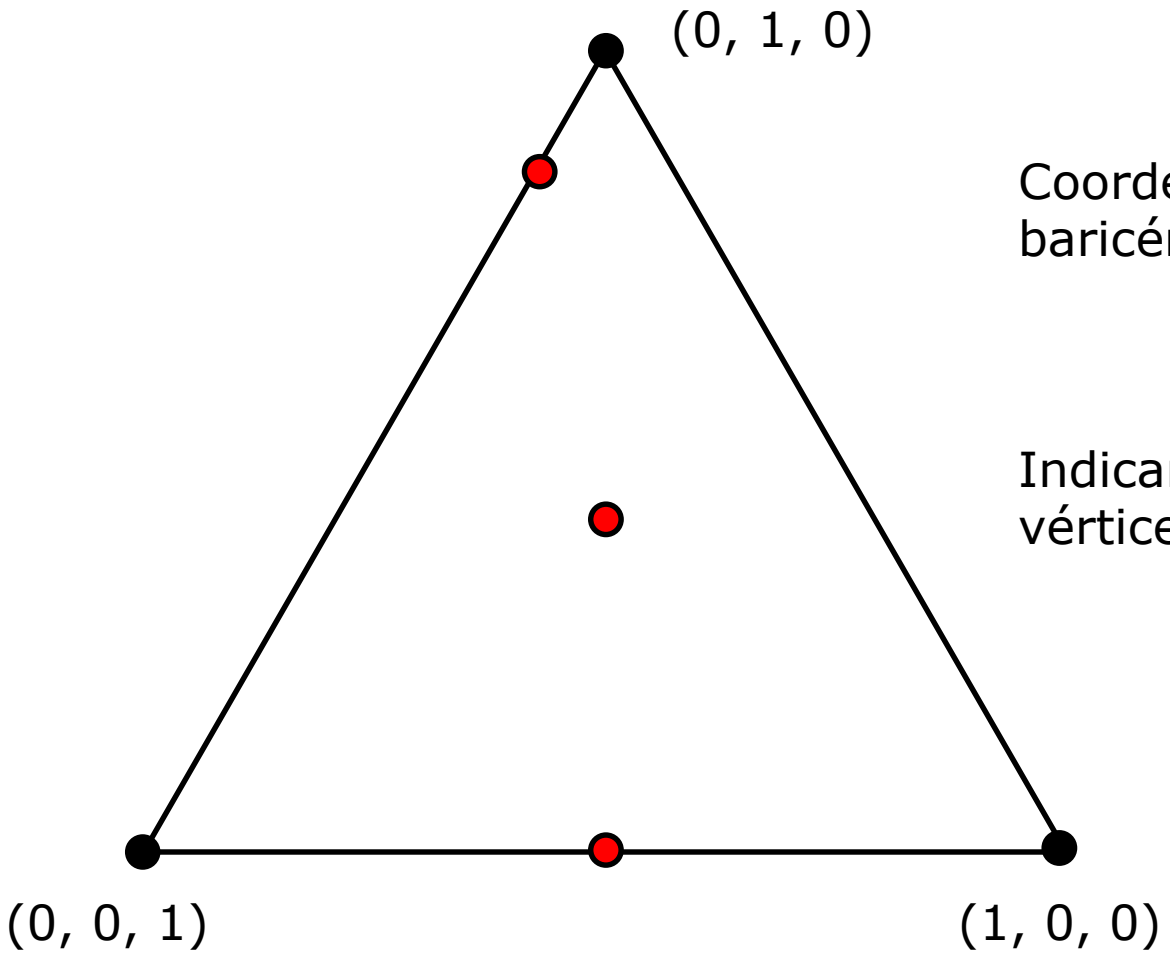
Teselación de cuadriláteros

Espacio paramétrico (u,v)



Teselación de triángulos

Espacio paramétrico (u, v, w)

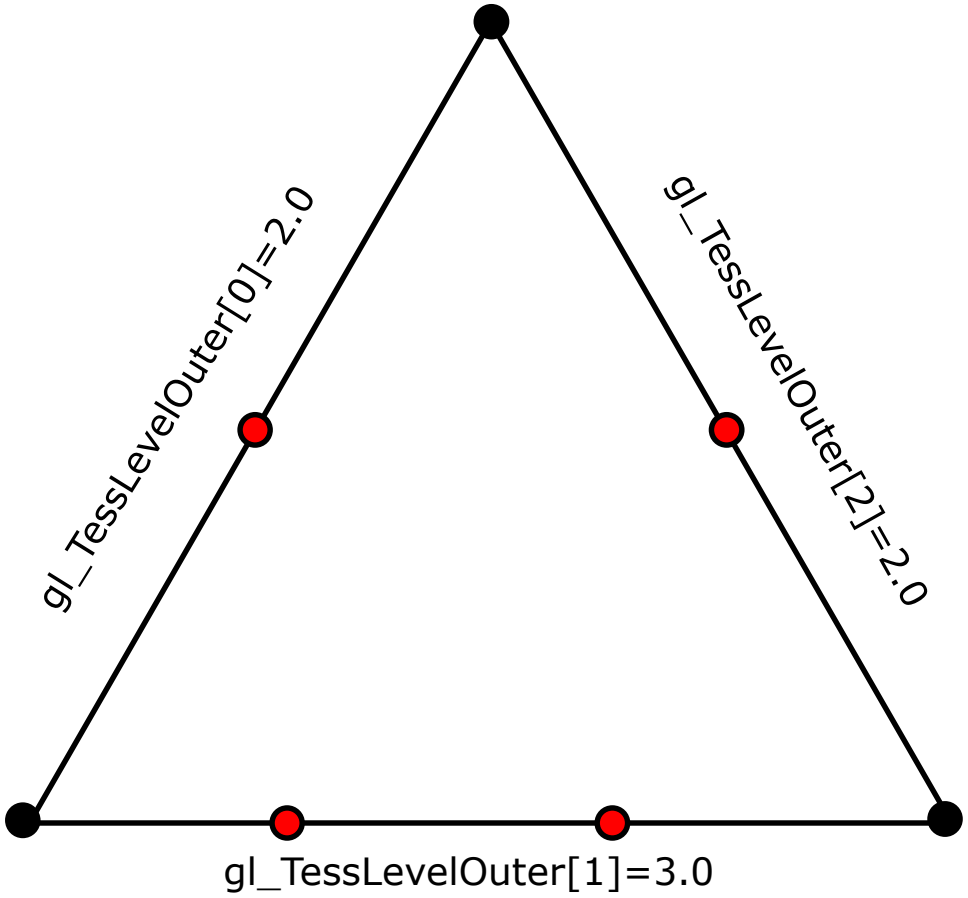


Coordenadas baricéntricas:

$$u+v+w = 1$$

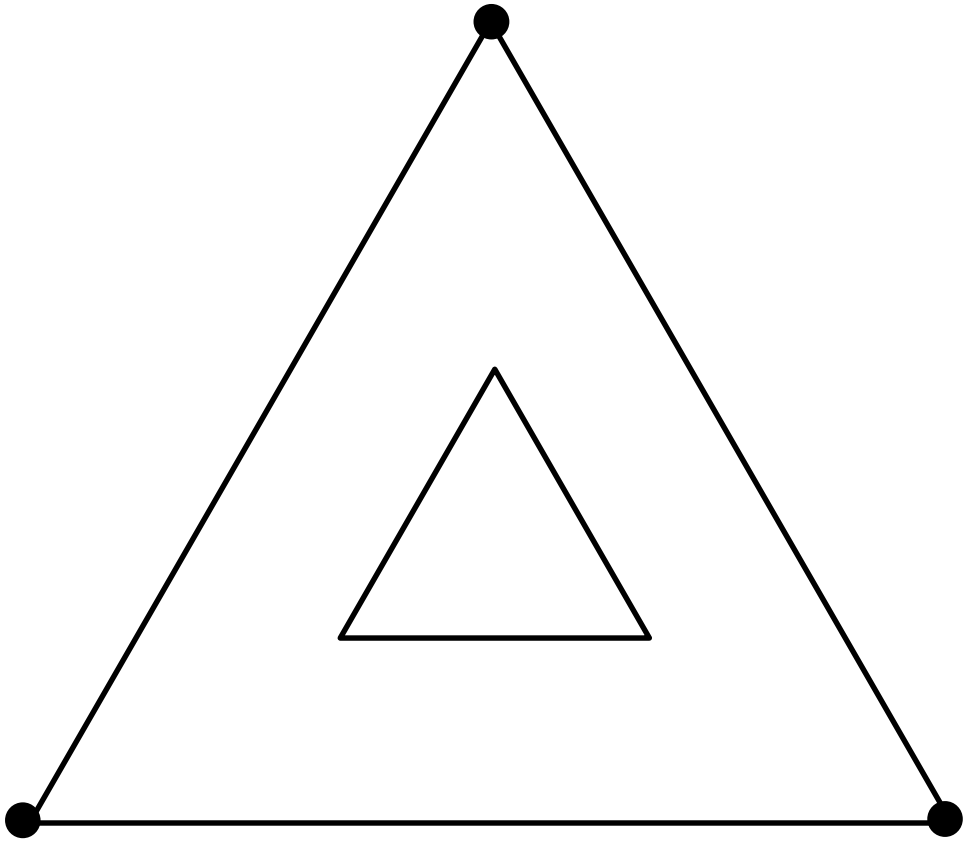
Indican proximidad a cada vértice.

Teselación de triángulos
Espacio paramétrico (u, v, w)



Teselación de triángulos

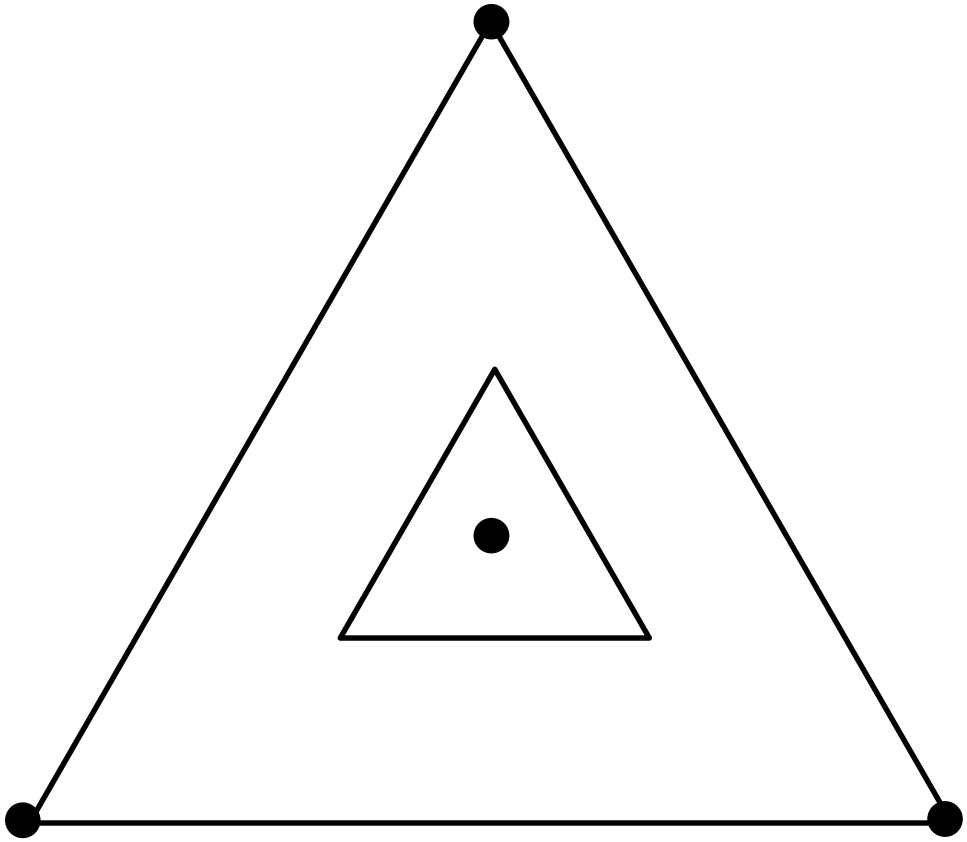
Espacio paramétrico (u, v, w)



gl_TessLevelInner[0]=3.0

Teselación de triángulos

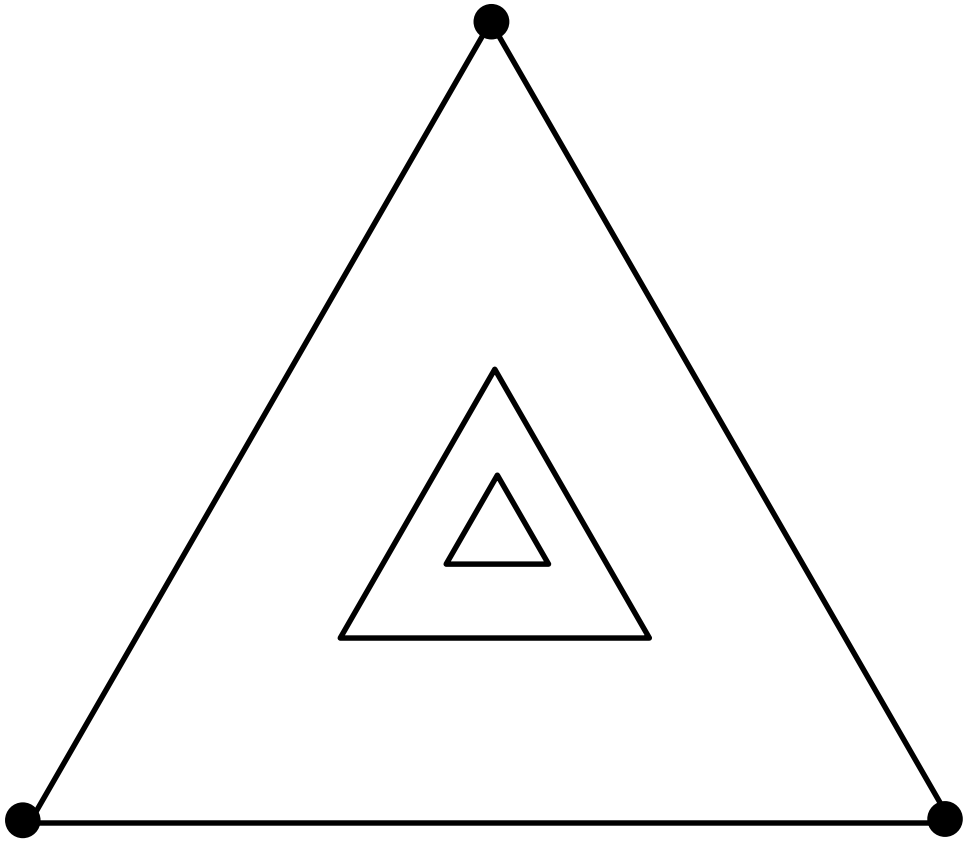
Espacio paramétrico (u, v, w)



gl_TessLevelInner[0]=4.0

Teselación de triángulos

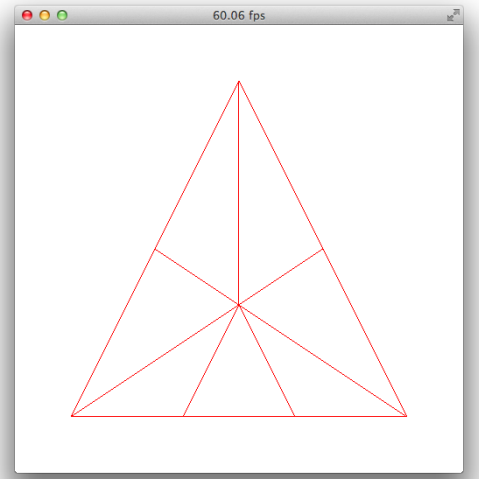
Espacio paramétrico (u, v, w)



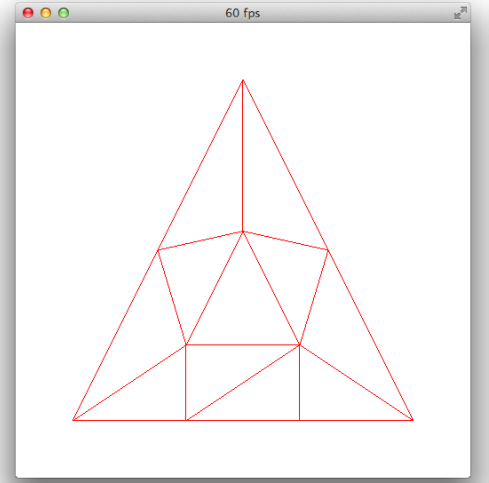
gl_TessLevelInner[0]=5.0

Teselación de triángulos

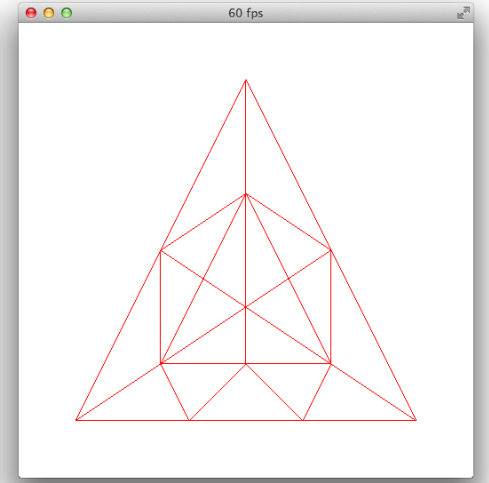
Espacio paramétrico (u, v, w)



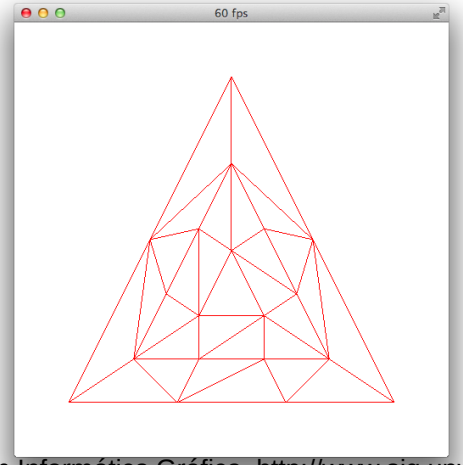
1,2



3



4



5

Ejemplos de `gl_TessLevelInner[0]`

ej7-2

Shaders de evaluación de teselación

- Se ejecuta una vez por cada coordenada de teselación generada por el teselador
  - Cuidado con niveles de teselación muy altos y shaders de evaluación complejos
- Se encarga de calcular la posición del vértice (y resto de atributos) asociada a la coordenada de teselación

Shaders de evaluación de teselación

- El primer paso es configurar el proceso de teselación con la instrucción `layout`:
  - `layout (opts) in;`
  - donde `opts` puede ser:
    - Tipo de teselación: `quads`, `triangles`, `isolines`
    - `cw`, `ccw`
    - espaciado: `equal_spacing`, `fractional_even_spacing`, `fractional_odd_spacing`
    - `point_mode`: el teselador generará puntos en vez de líneas o triángulos

Shaders de evaluación de teselación

- Variables predefinidas:
  - `in int gl_PatchVerticesIn;`
  - `in int gl_PrimitiveID;`
  - `in vec3 gl_TessCoord;`
  - `patch in float gl_TessLevelOuter[4];`
  - `patch in float gl_TessLevelInner[2];`

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

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

Shaders de evaluación de teselación
Espaciado

- Para un nivel de teselación *x* (real)

`equal_spacing`
1. Truncar *x* a [1..max]
2. Redondear al entero superior *n*
3. Dividir la arista en *n* segmentos iguales

`fractional_even_spacing`
1. Truncar *x* a [2..max]
2. Redondear al entero par superior, *n*
3. Dividir la arista en *n-2* segmentos iguales, y otros dos más pequeños, simétricos

`fractional_odd_spacing`
1. Truncar *x* a [1..max-1]
2. Redondear al entero impar superior, *n*
3. Dividir la arista en *n-2* segmentos iguales, y otros dos más pequeños, simétricos

Ejemplos: factor de teselación 4.5. Ver ejemplo ej7-3

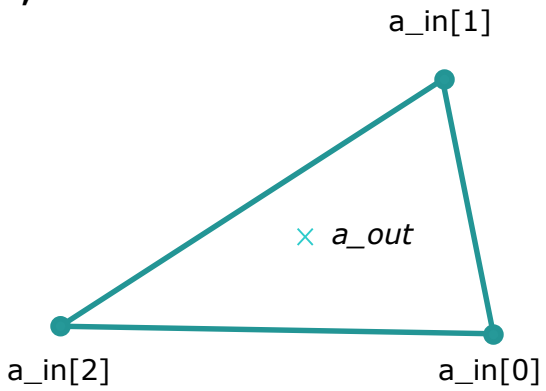


Interpolando atributos según las coordenadas de teselación

- Dados los atributos de los vértices originales, ¿cómo interpolar sus valores en los nuevos vértices?
- Para triángulos:

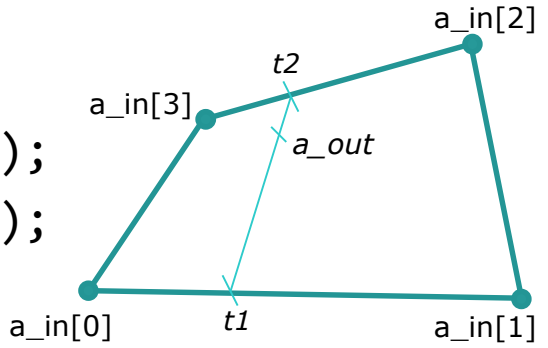
```
a_out = gl_TessCoord.x * a_in[0] +
        gl_TessCoord.y * a_in[1] +
        gl_TessCoord.z * a_in[2];
```

  - donde `a_in` puede ser un escalar o un vector



- Para cuadriláteros e isolíneas, interpolación bilineal:

```
T t1 = mix(a_in[0], a_in[1], gl_TessCoord.x);
T t2 = mix(a_in[3], a_in[2], gl_TessCoord.x);
a_out = mix(t1, t2, gl_TessCoord.y);
```



Ejemplo

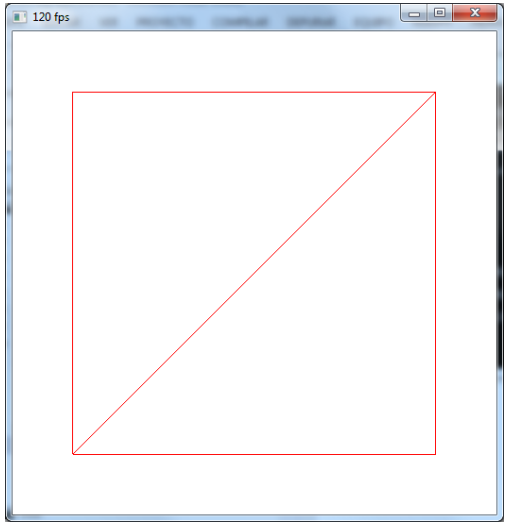
ej7-1

shader.vert

```
#version 410 core

in vec4 position;

void main()
{
  gl_Position = position;
}
```

shader.frag

```
#version 410 core

in vec4 color;
out vec4 finalColor;


void main() {
  finalColor = color;
}
```



4 vértices, dibujando un triangle fan

Ejemplo

ej7-1

main.cpp

```
void MyRender::setup() {
 […]
 auto m = std::make_shared<Mesh>();
 std::vector<glm::vec3> vs;
 vs.push_back(glm::vec3(-0.75, -0.75, 0.0));
 vs.push_back(glm::vec3(0.75, -0.75, 0.0));
 vs.push_back(glm::vec3(0.75, 0.75, 0.0));
 vs.push_back(glm::vec3(-0.75, 0.75, 0.0));
 m->addVertices(vs);

 DrawArrays *dc = new DrawArrays(GL_PATCHES, 0, 4);
 dc->setVerticesPerPatch(4);
 m->addDrawCommand(dc);

 model.addMesh(m);
[…]
```

Ejemplo

ej7-1

shader.tesc

```
#version 410 core

layout (vertices = 4) out;

void main() {
  gl_out[gl_InvocationID].gl_Position =
    gl_in[gl_InvocationID].gl_Position;
  if (gl_InvocationID == 0) {
    gl_TessLevelOuter[0] = 2.0;
    gl_TessLevelOuter[1] = 3.0;
    gl_TessLevelOuter[2] = 2.0;
    gl_TessLevelOuter[3] = 4.0;

    gl_TessLevelInner[0] = 3.0;
    gl_TessLevelInner[1] = 4.0;
  }
}
```

Ejemplo

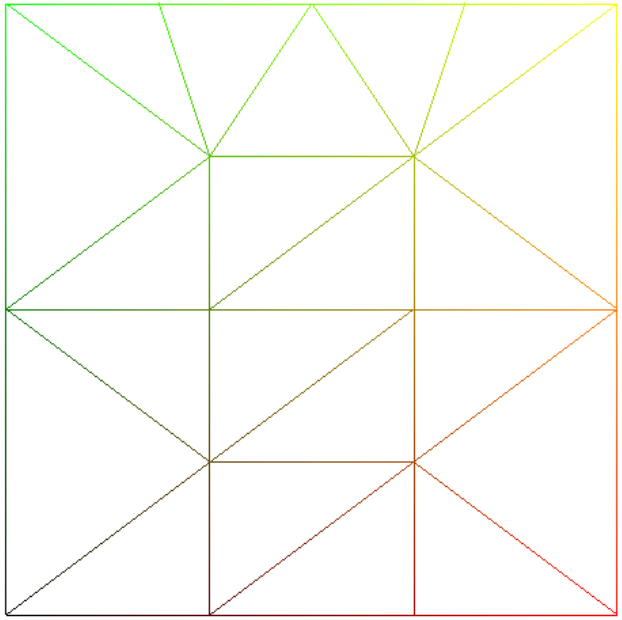
ej7-1

shader.tese

```
#version 410 core

layout (quads, equal_spacing, ccw) in;

out vec4 color;

void main() {
  color = vec4(gl_TessCoord, 1.0);

  vec4 p1 = mix(gl_in[0].gl_Position, gl_in[1].gl_Position,
                                      gl_TessCoord.x);
  vec4 p2 = mix(gl_in[3].gl_Position, gl_in[2].gl_Position,
                                      gl_TessCoord.x);

  gl_Position = mix(p1, p2, gl_TessCoord.y);
}
```

Ejemplo

ej7-1

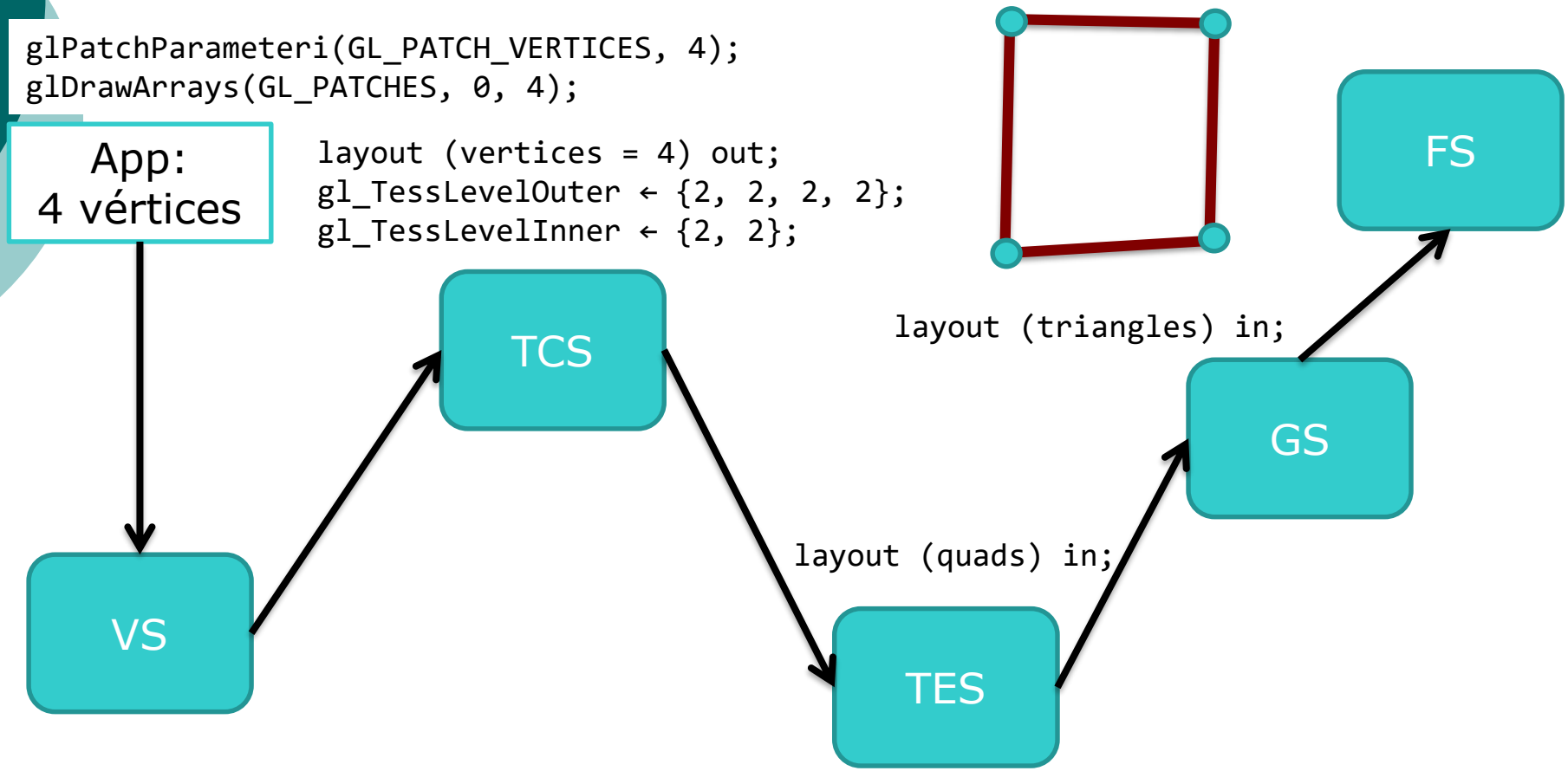


```
gl_TessLevelOuter[0] = 2.0;
gl_TessLevelOuter[1] = 3.0;
gl_TessLevelOuter[2] = 2.0;
gl_TessLevelOuter[3] = 4.0;

gl_TessLevelInner[0] = 3.0;
gl_TessLevelInner[1] = 4.0;
```

4 vértices, dibujando un parche

Resumen.

¿Cuántas veces se ejecuta cada shader?

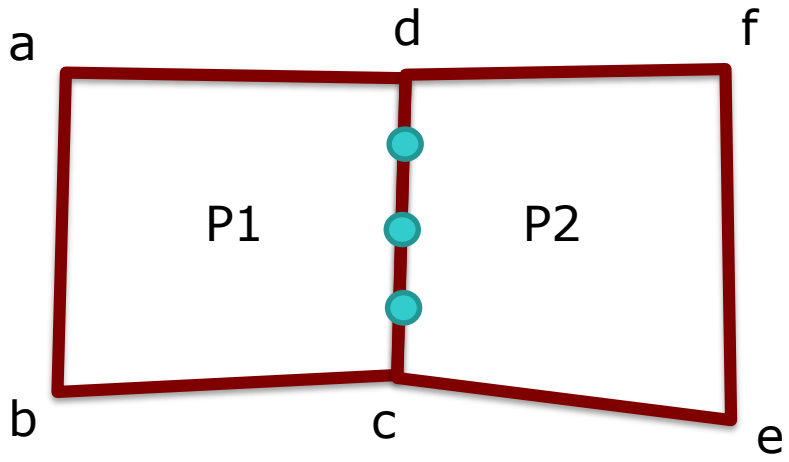


Agujeros en la malla teselada

- Es común que una malla esté compuesta por varios parches contiguos
- Si las aristas comunes no se teselan exactamente de la misma forma, es posible que aparezcan agujeros en la malla
  - Solución: calcula cada nivel de teselación externo a partir de la posición de los dos vértices de su arista correspondiente

Agujeros en la malla teselada

- Aparte de usar los mismos factores de teselación en las aristas compartidas, está el problema de la precisión de la aritmética en coma flotante



Usar el calificador precise para asegurar que ambos cálculos generan el mismo resultado

Generando geometría

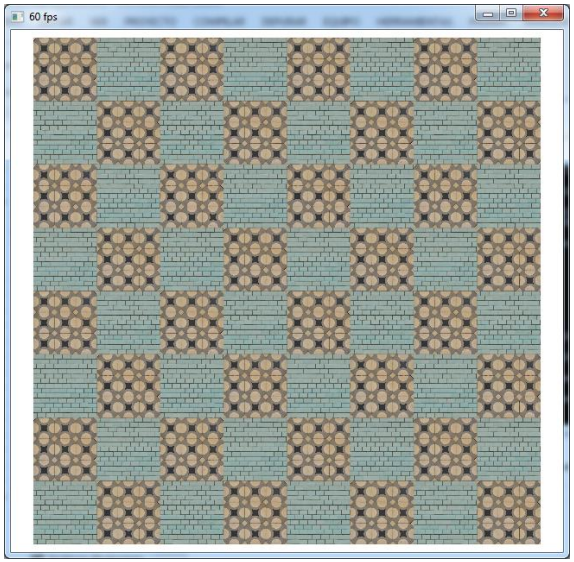
Dibujo instanciado

- Son funciones de dibujo que permiten dibujar la misma primitiva un número de veces especificado por parámetro (siempre la misma primitiva)
- Las funciones equivalentes a glDrawArrays y glDrawElements son:
  - `void glDrawArraysInstanced(mode, first, count, primcount)`
  - `void glDrawElementsInstanced(mode, count, type, indices, primcount)`
  - donde
    - `primcount` es el número de instancias (copias) a realizar
    - El resto de parámetros tienen el mismo significado que antes (ver Tema 1)

Generando geometría

Dibujo instanciado

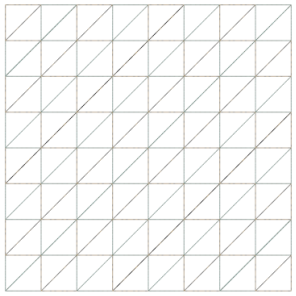
- Dentro del shader de vértice, la variable predefinida gl_InstanceID identifica la instancia que se está dibujando (desde *0 hasta primcount-1*)
- Dicha variable se puede utilizar para aplicar una transformación distinta a cada primitiva



ej7-4

Dibujado con:

```
glDrawArraysInstanced(GL_TRIANGLE_FAN, 0, 4,
                                        8 * 8)
```



Generando geometría

Dibujo instanciado

shader.vert

```
in vec3 position;
in vec2 texCoord;

out PerVertex {
 vec2 texCoordFrag;
 flat int whichTile;
};

void main()
{
 int x = gl_InstanceID & 7;
 int y = gl_InstanceID >> 3;
 whichTile = (x & 1) ^ (y & 1);

 texCoordFrag = texCoord;
 vec3 newpos = position - vec3(float(x)-4.0+0.5, float(y)-4.0+0.5, 0.0);
 gl_Position = modelviewprojMatrix * vec4(newpos, 1.0) ;
}
```

Generando geometría

Dibujo instanciado

shader.frag

```
#version 330 core

uniform sampler2DArray texUnit;

in PerVertex {
  vec2 texCoordFrag;
  flat int whichTile;
} from_vs;

out vec4 fragColor;

void main()
{
  fragColor = texture(texUnit,
        vec3(from_vs.texCoordFrag, from_vs.whichTile));
}
```