Universidad de Huelva

Tema 3

Las etapas de renderizado

Índice

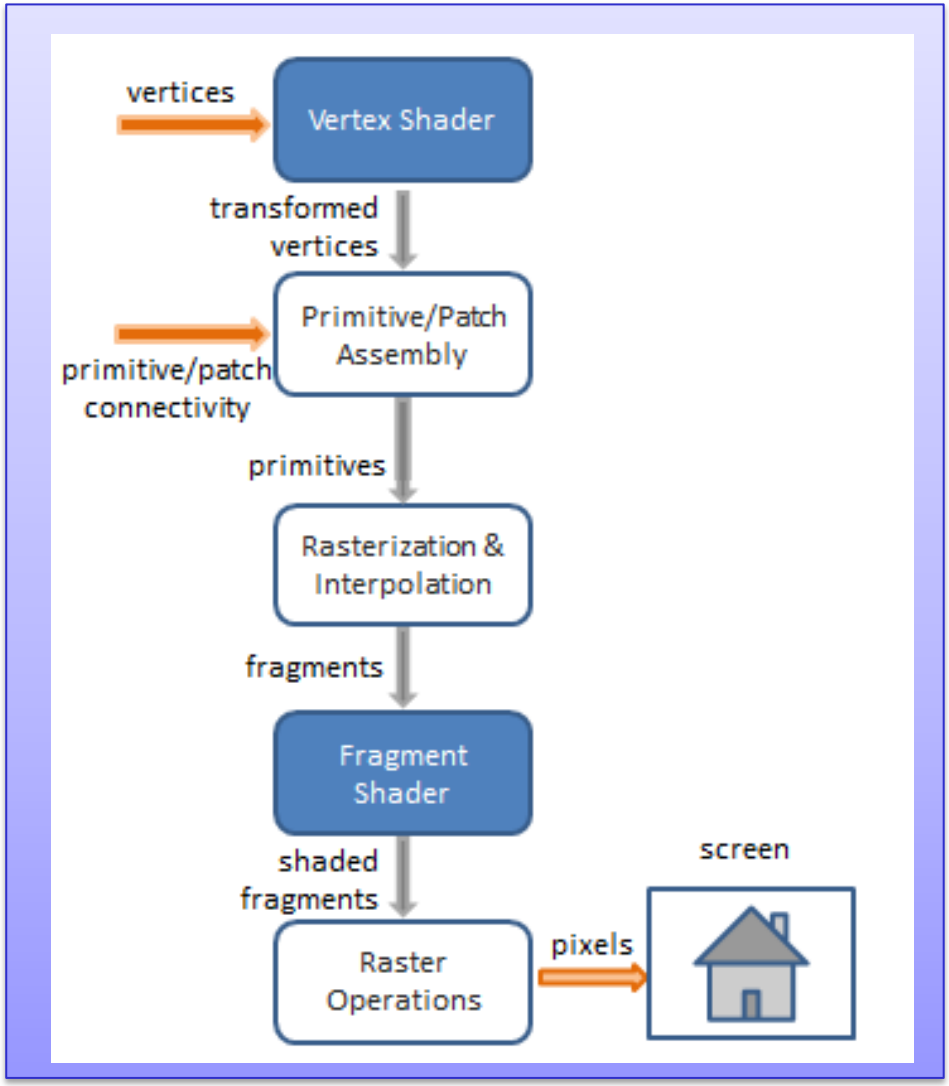
- Se conoce como **pipeline de renderizado** al conjunto de procesos que se ejecutan en la GPU (tarjeta gráfica) para generar una imagen.
- El esquema tradicional de funcionamiento de OpenGL consistía en una serie de etapas de renderizado que podían configurarse activando determinadas opciones por medio del comando glEnable(…) y asignando ciertos parámetros mediante comandos específicos.
- En este esquema, el funcionamiento de cada etapa de renderizado estaba fijado por el estándar. A medida que se incorporaban más características a OpenGL, la función a ejecutar en cada etapa se iba complicando con más y más opciones.

- En el año 2001 aparecen las primeras tarjetas gráficas en las que se incluye la posibilidad de programar externamente algunas etapas del pipeline (NVIDIA GeForce 3, ATI Raedon 8500).
- Las etapas programables eran *vertex_program*, que se ejecutaba sobre cada vértice, y *fragment_program*, que se ejecutaba sobre cada pixel de cada fragmento.
- Inicialmente, esta programación debía hacerse en un ensamblador específico y con un tamaño de programa reducido. Pese a las limitaciones, estas etapas programables permitían incorporar nuevas técnicas de interpolación, mezcla o sombreado sin necesidad de añadirlas como opciones del estándar OpenGL.

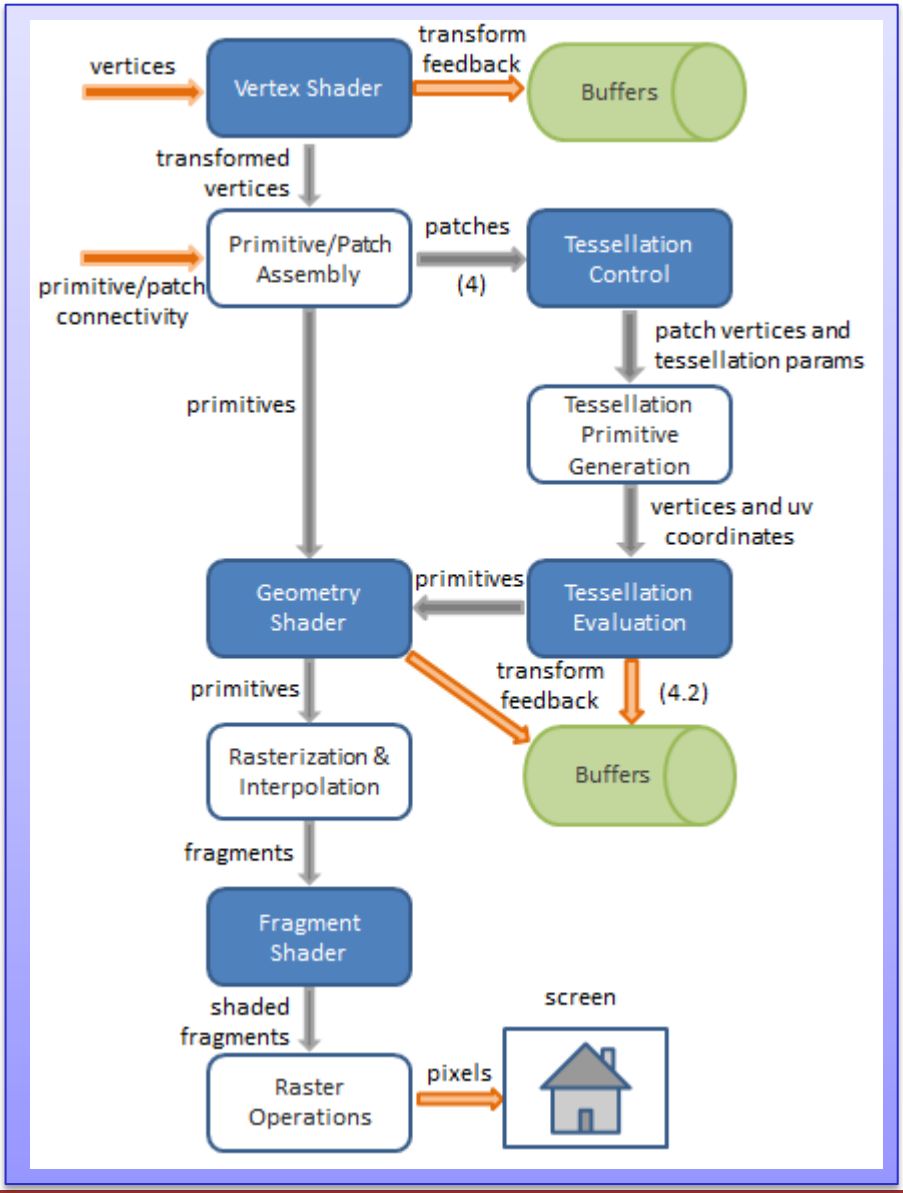
- A partir de 2003, las tarjetas gráficas mejoraron notablemente las capacidades de las etapas programables (NVIDIA GeForce FX, ATI Radeon 9700, 3DLab WildCat VP)
- De esta forma, las etapas programables (*Vertex_shader* y *Fragment_shader*) pudieron programarse con lenguajes de alto nivel que permiten introducir instrucciones más complejas. Esto llevó a la definición del lenguaje GLSL (OpenGL Shading Language) dentro del estándar OpenGL.
- Las versiones sucesivas fueron ampliando las capacidades de programación, incluyendo más tipos de datos, más funciones predefinidas (*built-in functions*) y nuevas etapas: *Geometry_shader*, *Tesselation_control_shader* y *Tesselation_evaluation_shader*.

- Etapas de renderizado:
  - El pipeline actual está formado por varias etapas, algunas de ellas programables y otras con funcionalidad fija.
  - Cada etapa supone la ejecución de su función un gran número de veces (por cada vértice, por cada primitiva, por cada píxel).
  - Las tarjetas gráficas están formadas por un gran número de unidades de proceso (cores) lo que permite ejecutar en paralelo todas estas funciones.
  - Por ejemplo, la última generación de tarjetas gráficas de Nvidia (arquitectura Ampere) incluyen hasta 10496 núcleos CUDA (enlace)

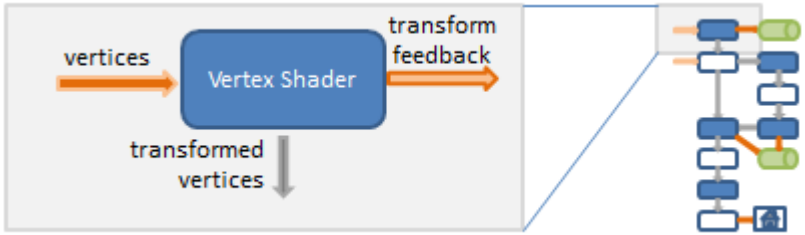
- Pipeline básico:



- Pipeline completo:



Índice



- Se ejecuta sobre cada vértice por separado, sin tener en cuenta la existencia de otros vértices ni la primitiva a la que pueda pertenecer.
- Utiliza atributos de entrada asociados a cada vértice. La posición, color, normal, coordenadas de textura se tratan como atributos, pero se pueden definir atributos libremente.
- Utiliza variables uniformes (*uniforms*) para compartir valores de entrada entre los vértices. Las matrices de transformación de sistemas de coordenadas se tratan de esta forma.

- Además, utiliza dos entradas predefinidas:

```
in  int   gl_VertexID;
in  int   gl_InstanceID;
```

- El atributo **gl_VertexID** contiene el índice del vértice en los arrays de atributos.
- El atributo **gl_InstanceID** contiene el número de instancia del vértice. Los comandos de dibujo (*glDraw*) pueden lanzar varias instancias de los vértices. Si no se usan instancias, el valor de gl_InstanceID es 0.
- Genera como salida la posición del vértice (**gl_Position**) en coordenadas normalizadas (clipping volume) y otros valores que se deseen trasladar a otras etapas.

- Los atributos de los vértices se configuran por medio de Vertex Array Objects (VAO) y se almacenan en la memoria de la GPU por medio de Vertex Buffer Objects (VBO).
- Para definir los VAOs se utilizan las siguientes funciones

```
// crea n VAOs y almacena sus identificadores en el array.
void glGenVertexArrays(GLsizei n, GLuint *vaoIDs);

// elimina n VAOs referenciados en los elementos del array.
void glDeleteVertexArrays(GLsizei n, const GLuint *vaoIDs);

// verifica si un identificador corresponde a un VAO
GLboolean glIsVertexArray(GLuint vaoID);

// activa un determinado VAO
void glBindVertexArray(Gluint vaoID);
```

- Para definir un almacén de datos en la memoria de la tarjeta gráfica se utiliza un VBO. El campo target indica el tipo de buffer (GL_ARRAY_BUFFER para los VBO).

```
// crea n VBOs y almacena sus identificadores en el array.
void glGenBuffers(GLsizei n, GLuint * vboIDs);

// elimina n VBOs referenciados en los elementos del array.
void glDeleteBuffers(GLsizei n, const GLuint *vboIDs);

// verifica si un identificador corresponde a un VBO
GLboolean glIsBuffer(GLuint vboID);

// activa un determinado VBO
void glBindBuffer(GLenum target, GLuint vboID);

// almacena los datos en el VBO activo
void glBufferData(GLenum target, GLsizeiptr size,
       const GLvoid * data, GLenum usage);
```

- Para definir un atributo perteneciente a un VAO se usa

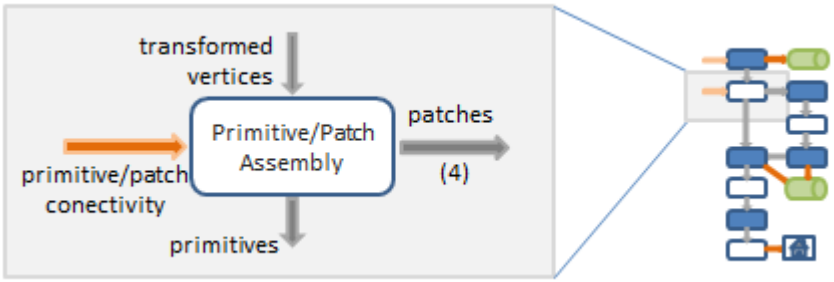
```
void glVertexAttribPointer(GLuint index,
        GLint size,
        GLenum type,
        GLboolean normalized,
        GLsizei stride,
        const GLvoid * pointer);
```

donde *index* es el índice del atributo en el VAO, *size* es el número de datos por cada atributo (1,2,3 o 4), *type* es el tipo de dato, *normalized* indica si los valores deben normalizarse o no, *stride* es el offset (número de bytes) entre los valores de dos atributos consecutivos y *pointer* es el offset del primer atributo respecto del comienzo del buffer.

- Los datos se toman desde el buffer activo (glBindBuffer).

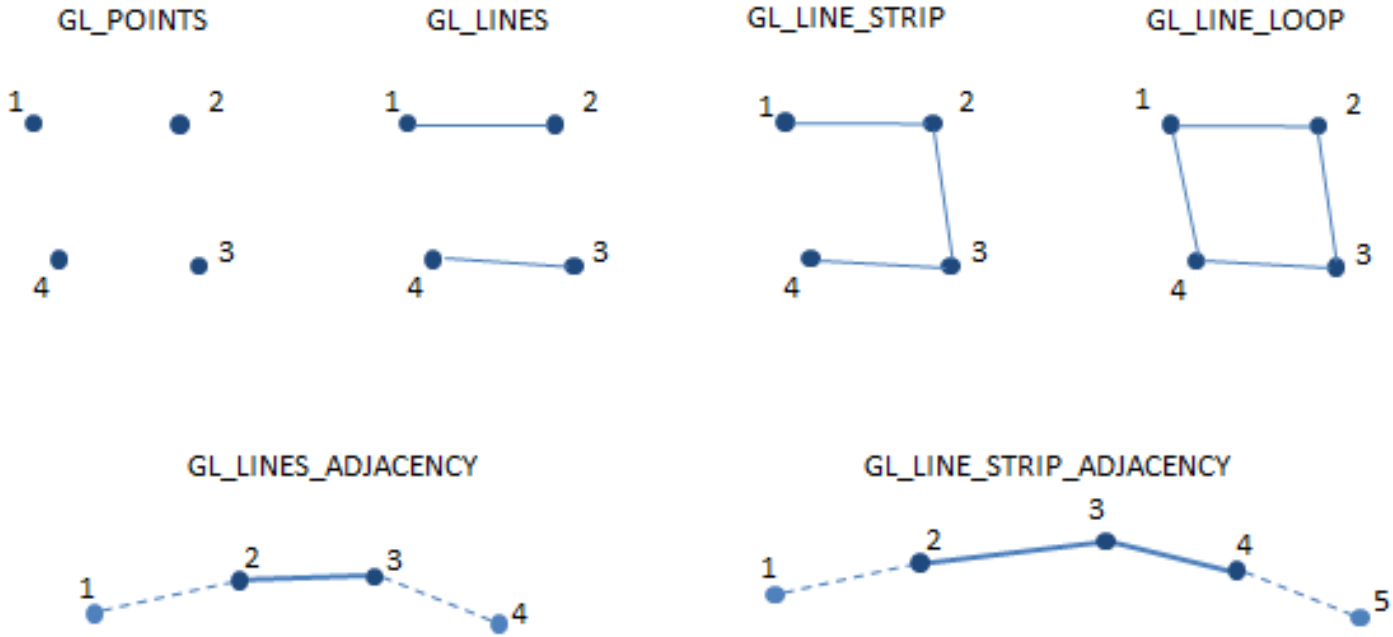
- Ejemplo de programación de un Vertex shader

```
#version 410

uniform mat4 projModelViewMatrix;
uniform mat3 normalMatrix;

in vec3 positionV;
in vec3 normalV;
in vec2 texCoordV;

out vec2 texCoord;
out vec3 normal;

void main()
{
    texCoord = texCoordV;
    normal = normalize(normalMatrix * normalV);
    gl_Position = projModelViewMatrix * vec4(positionV, 1.0);
}
```



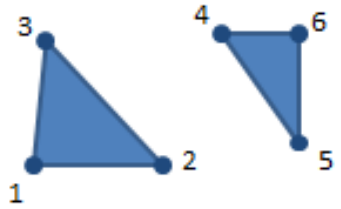
- Se trata de una etapa no programable.
- Recibe como entrada la indicación de las primitivas a dibujar (definidas en el comando *glDraw*) y los atributos de los vértices de la primitiva generados por el *Vertex_shader*.
- Las primitivas pueden ser puntos, líneas, triángulos o parches (*patches*). Los parches son un tipo de primitiva especial que se utiliza como entrada para los *shaders* de teselado. Estos *shaders* convierten la información de los parches en un conjunto de primitivas geométricas.

- Primitivas admitidas como puntos o líneas.
  - Las primitivas definidas como *adjacency* deben ser tratadas por el *shader* de geometría. El resto pueden ser tratadas directamente por el *Fragment_shader*.

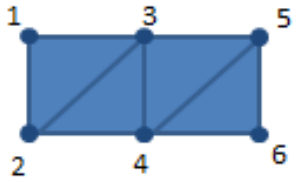


- Primitivas de tipo triangular

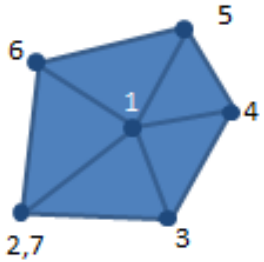
GL_TRIANGLES



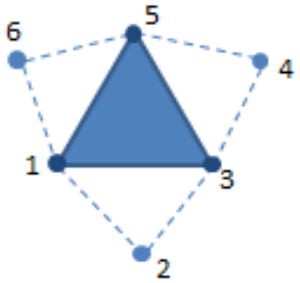
GL_TRIANGLE_STRIP



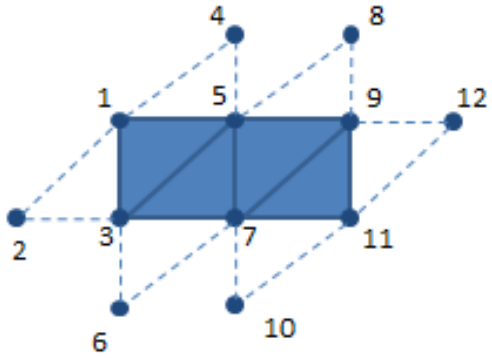
GL_TRIANGLE_FAN



GL_TRIANGLES_ADJACENCY



GL_TRIANGLE_STRIP_ADJACENCY



- Primitivas
  - Para lanzar un proceso de renderizado, asumiendo que el programa con los shaders está en uso y que los datos de los vértices y de las variables uniformes ya han sido cargados, se utilizan los comandos de dibujo *glDraw…()*.

```
void glDrawArrays( GLenum mode,
       GLint first,
       GLsizei count);

void glDrawElements(   GLenum mode,
       GLsizei count,
       GLenum type,
       const GLvoid * indices);
```

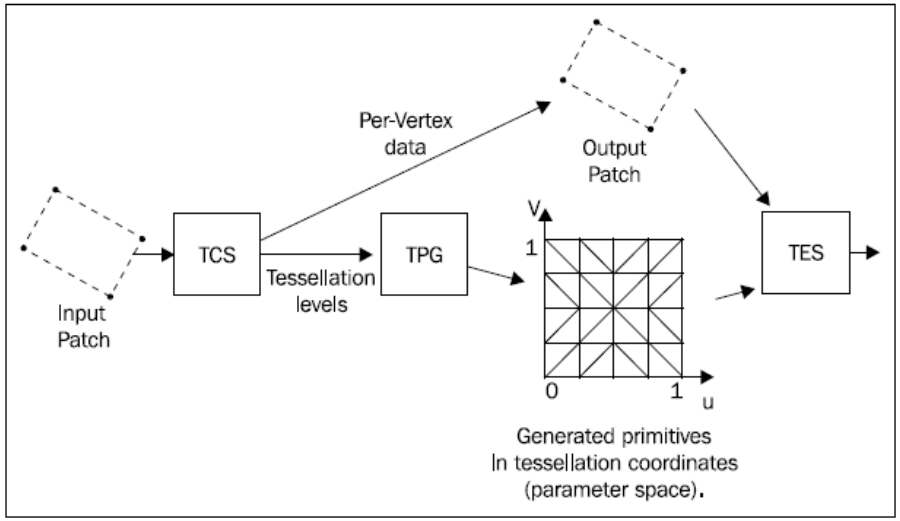
- Primitivas
  - El comando *glDrawArrays* asigna los vértices a las primitivas de manera consecutiva.
  - El comando *glDrawElements* asigna los vértices a partir de una lista de índices que toma del buffer GL_ELEMENTS_ARRAY_BUFFER activo.
  - Las versiones instanciadas de estos comandos ejecutan las primitivas un cierto número de veces (*primcount*).

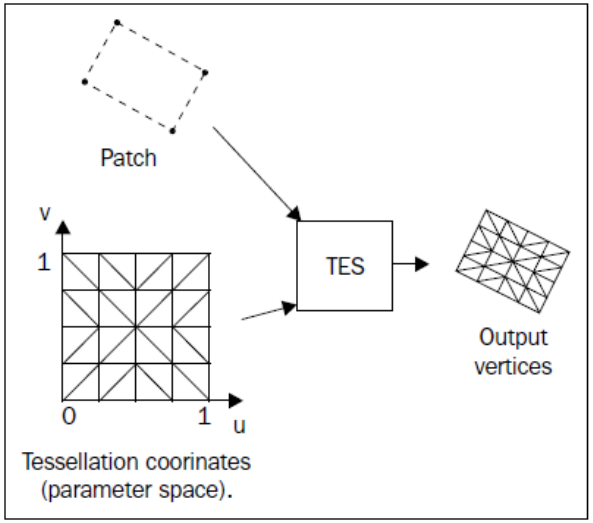
```
void glDrawArraysInstanced( GLenum mode,
       GLint first,
       GLsizei count,
       GLsizei primcount);

void glDrawElementsInstanced( GLenum mode,
       GLsizei count,
       GLenum type,
       const void * indices,
       GLsizei primcount);
```

- Se trata de una etapa del pipeline que recibe como entrada los datos de un parche y genera como salida nuevas primitivas geométricas que pueden ser puntos, líneas o triángulos.
- Está dividida en tres partes: control de teselado, generación de primitivas y evaluación de teselado. La generación de primitivas tiene una funcionalidad fija. El control y la evaluación son etapas programables, lo que da lugar al *Tessellation_control_shader* y al *Tessellation_evaluation_shader*.
- El teselado toma como entrada un array de vértices y genera como salida otro array de nuevos vértices.

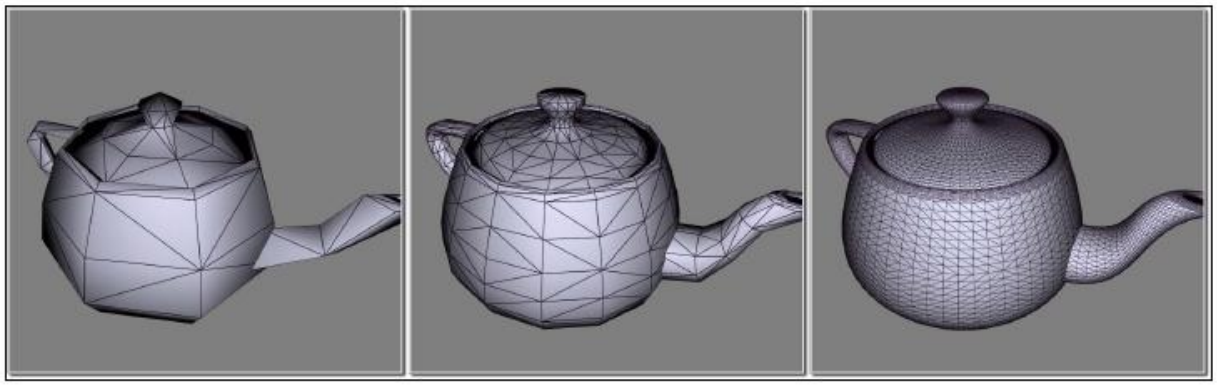
- El TCS recibe como entrada los vértices del patch y genera como salida la configuración de la siguiente etapa y los valores de los vértices de las primitivas a crear
- El TPG genera las primitivas en un sistema de coordenadas denominado espacio de parámetros. Las coordenadas en este espacio se denominan (u,v). El espacio de parámetros está definido en el rango [0,1] de cada coordenada.
- El TES transforma las coordenadas del espacio de parámetros en coordenadas de posición de los vértices.

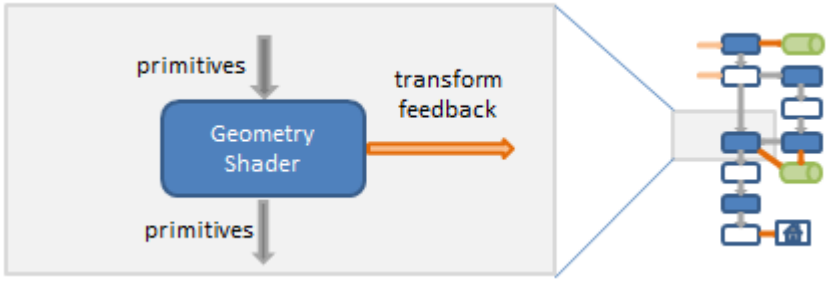
- Esquema de funcionamiento:





- El teselado puede utilizarse, por ejemplo, para dibujar objetos por medio de superficies de Bezier. De esta forma es posible dibujar el objeto con diferentes niveles de detalle.
- Por ejemplo, esta tetera se ha descrito por medio de superficies de Bezier. Los dibujos utilizan los niveles de teselado 2 (izquierda), 4 (centro) y 16 (derecha).





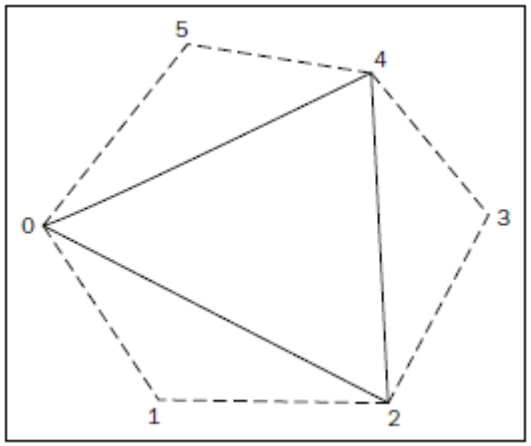
- Se trata de una etapa opcional.
- Recibe como entrada las primitivas ensambladas (incluidas primitivas con información de vértices adyacentes) y permite hacer modificaciones, incluyendo la generación de nuevas primitivas.
- Las primitivas de salida pueden ser puntos, líneas o triángulos. La información "adyacente" debe tratarse dentro de este shader.
- Esta etapa puede duplicar primitivas, crear primitivas nuevas y dirigir primitivas a distintos layers, por ejemplo.

- Una limitación importante es que todas las primitivas que genere 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. Por ejemplo, sustituir puntos por pequeñas imágenes (point sprites).

- 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.

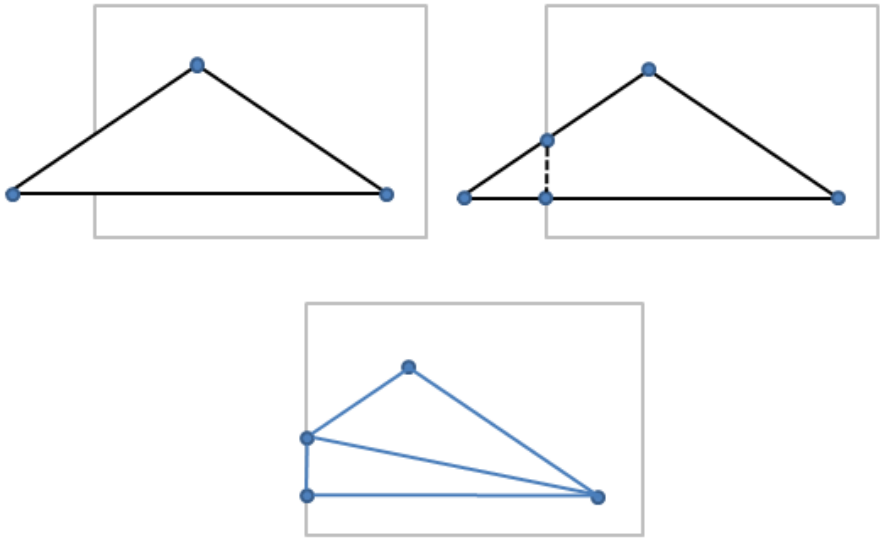
- El shader de geometría tiene como entrada la variable ***gl_in*[]**. Se trata de un array que almacena la información de cada vértice de la primitiva ensamblada.
- La estructura gl_in contiene los siguientes campos:
  - gl_Position: posición del vértice
  - gl_PointSize: tamaño de punto del vértice
  - 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.

- Ejemplo de uso: Dibujar siluetas.
  - Se basa en el uso de 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.
  - El shader de geometría estudia si los triángulos adyacentes se ven por su cara posterior y genera primitivas en la posición de las aristas.



- Se trata de una etapa no programable.
- La entrada está formada por las primitivas ensambladas, que referencian a los vértices cuyos atributos se calcularon en el Vertex_shader.
- El primer paso es estudiar que vértices quedan dentro del clipping volume y cuales fuera.

- Las primitivas que quedan fuera del clipping volume son desechadas. Las que quedan parcialmente fuera deben adaptarse creando nuevos vértices sobre el borde del clipping volume.

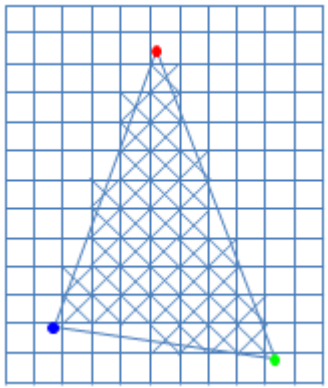


- Los atributos de los nuevos vértices se obtienen por interpolación lineal, salvo los atributos declarados "flat".
- Cuando un atributo se declara "flat" su valor se copia a todos los vértices de la primitiva. Típicamente, el valor utilizado es el del último vértice, aunque esto puede configurarse con el comando

glProvokingVertex(enum provokeMode);

- donde *provokeMode* puede ser GL_FIRST_VERTEX_CONVENTION o GL_LAST_VERTEX_CONVENTION.

- La rasterización consiste en identificar que pixels forman parte de la primitiva a dibujar.
- El primer paso es calcular la orientación de la primitiva (frontal o posterior). Si la opción de FaceCulling está activa hay que desechar las primitivas con orientación incorrecta.
- El siguiente paso es estudiar que pixels están dentro de la primitiva. Para ello se estudia si el centro del pixel se encuentra dentro del triángulo que forman los vértices.

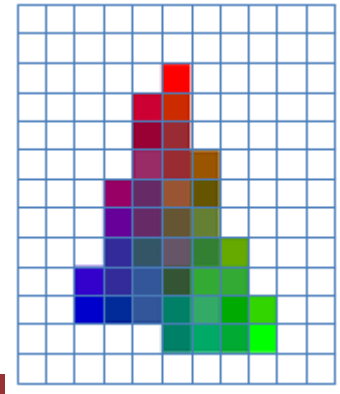


- A continuación se calculan los valores de los atributos de cada pixel. Para ello se calculan los coeficientes de interpolación del pixel (*a*,*b*,*c*), considerando las coordenadas del centro del pixel (*p*) y las de los vértices (*pa*, *pb* y *pc*).

$$p = a * pa + b * pb + c*pc$$

donde a+b+c = 1 y están definidos entre 0 y 1.

- Los atributos del pixel se obtienen por interpolación de los atributos de los vértices por medio de los coeficientes calculados.



- Es una etapa programable y no es opcional.
- Se ejecuta por cada fragmento, es decir, por cada pixel de cada primitiva.
- Recibe como entrada los valores interpolados para el pixel.
- Además, puede utilizar las entradas predefinidas: *gl_FragCoord*, *gl_FrontFacing* y *gl_PrimitiveID*.
- Debe generar como salida el color del pixel asociado. También puede generar salidas hacia otros buffers.

- Ejemplo de Fragment shader

```
#version 400

in vec4 ObjectColor;
out vec4 FragColor;

void main()
{
    FragColor = ObjectColor;
}
```

- Es una etapa no programable.
- Es la responsable de combinar los distintos fragmentos para generar la versión final de la imagen.
- Algunos de los procesos incluidos en esta etapa pueden ser ejecutados antes del Fragment shader por motivos de eficiencia.
- Procesos incluidos en esta etapa:
  - Pixel ownership test
  - Scissor test
  - Stencil test
  - Depth test
  - Blending

- Pixel ownership test:
  - Verifica que un pixel sea responsabilidad de OpenGL o no. Por ejemplo, cuando una ventana está parcialmente tapada por otra, este test detecta que los pixeles ocultos no hay que generarlos.
- Scissor test:
  - Permite definir el trozo de raster que se desea generar (por defecto se genera completo). Si un pixel queda fuera de la zona recortada, no se genera.
- Stencil test:
  - Compara el valor del campo stencil de un fragmento con el valor de stencil almacenado en un buffer y decide si genera el pixel o no.

- Depth test:
  - Compara la profundidad del pixel a generar con la profundidad almacenada en un buffer y decide si genera el pixel o no. Esto permite eliminar los pixels que corresponden a primitivas ocultas por otras primitivas más cercanas.
- Blending:
  - Permite mezclar el color del pixel a generar con el color del pixel que ya estuviera almacenado en el buffer de color. Esto permite tratar objetos traslúcidos (semitransparentes) utilizando colores que incluyan valores de la componente alpha (grado de transparencia). El formato de color utilizado en OpenGL es RGBA, que incluye el grado de transparencia.