Universidad de Huelva

Tema 9

Teselado

Índice

9.1 Los shaders de teselado

9.2 Teselado de una curva

9.3 Teselado de un cuadrilátero

9.4 Teselado de una superficie 3D

9.5 Teselado basado en profundidad

9.1 Los shaders de teselado

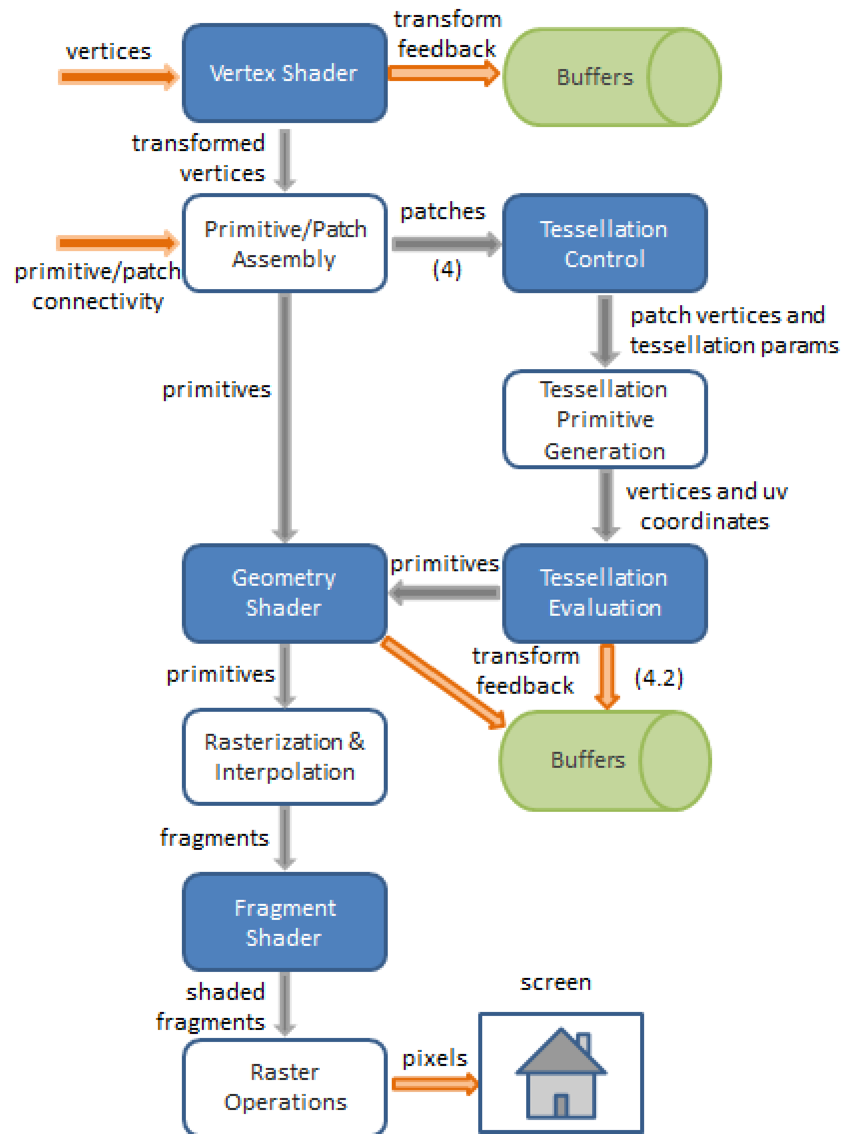
9.2 Teselado de una curva

9.3 Teselado de un cuadrilátero

9.4 Teselado de una superficie 3D

9.5 Teselado basado en profundidad

- Pipeline de renderizado



- La etapa de teselado es opcional.
- Cuando un programa contiene shaders de teselado, solo se pueden renderizar primitivas de tipo GL_PATCHES.
- Las primitivas *patch* no tienen un significado geométrico definido. No son más que una forma de agrupar la información que se le pasa al shader de control de teselado. Los vértices que se utilizan como miembros de la primitiva no tienen por qué tener un significado geométrico (puede que se utilize su posición o no a la hora de procesar las nuevas primitivas).
- El número de vértices incluido en las primitivas *patch* es configurable. El número máximo de vértices puede obtenerse así:

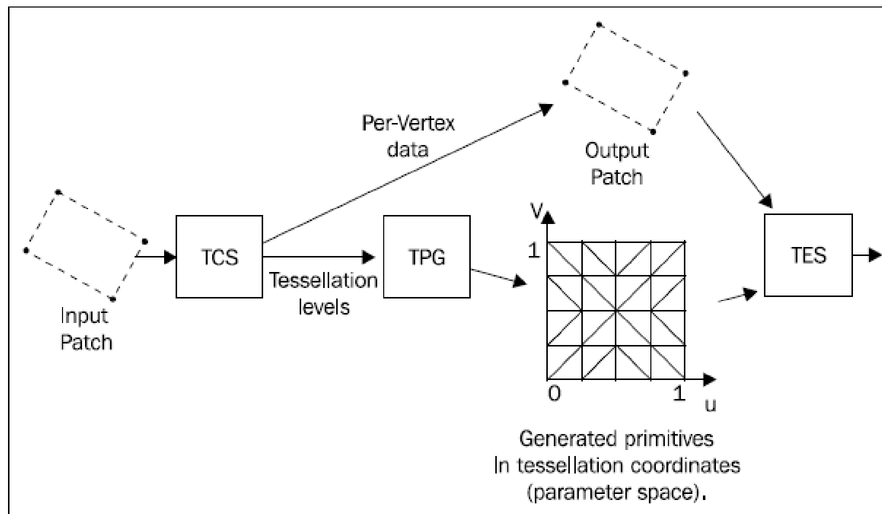
```
glGetIntegerv(GL_MAX_PATCH_VERTICES, &maxVerts);
```

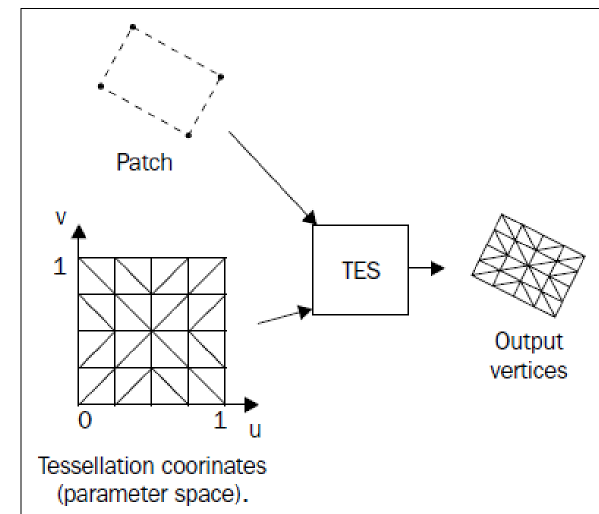
- Para definir el número de vértices de una primitiva patch se usa

  glPatchParameteri(GL_PATCH_VERTICES, numPatchVerts);

- La etapa de teselado se divide en tres fases:
  - TessellationControlShader (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.
  - TessellationPrimitiveGenerator (TPG): genera un conjunto de primitivas en base a la configuración recibida por medio de una serie de algoritmos predefinidos.
  - TessellationEvaluationShader (TES): genera los valores finales de los vértices de las primitivas creadas por el TPG utilizando la información de los vértices generada por el TCS.

- 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.





9.1 Los shaders de teselado

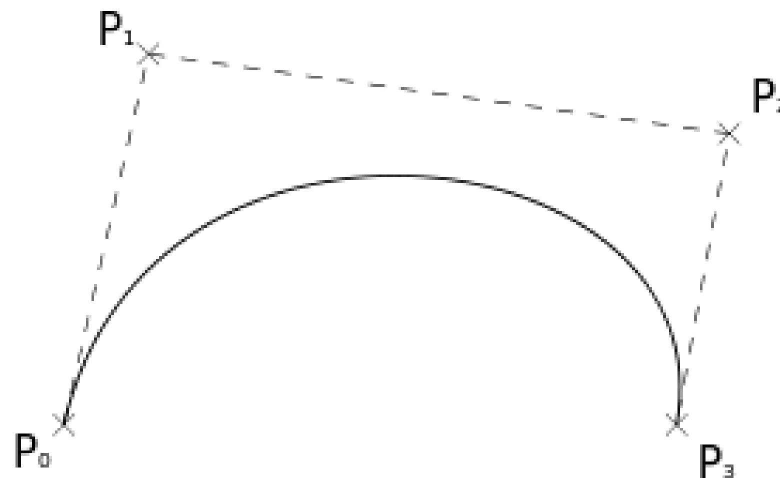
9.2 Teselado de una curva

9.3 Teselado de un cuadrilátero

9.4 Teselado de una superficie 3D

9.5 Teselado basado en profundidad

- Una curva de Bezier cúbica se describe por medio de 4 puntos. El primer punto y el último punto definen el comienzo y el final de la curva. Los dos puntos intermedios pueden encontrarse fuera de la curva y determinan la forma de doblarse de la curva.



- Las curvas de Bezier se basan en el uso de polinomios de Bernstein

$$B_i^n(t) = \frac{n!}{i!(n-i)!} \cdot (1-t)^{n-i} \cdot t^i$$

- Las curvas de Bezier de grado *n* se calculan como

$$P(t) = \sum_{i=0}^{n} B_i^n(t) \cdot P_i$$

donde P_i son las coordenadas del punto i-ésimo

- Para la curva de Bezier cúbia (n=3)

$$P(t) = B_0^3(t) \cdot P_0 + B_1^3(t) \cdot P_1 + B_2^3(t) \cdot P_2 + B_3^3(t) \cdot P_3$$

- En este caso, los polinomios de Bernstein quedan

$$B_0^3(t) = (1-t)^3$$
$$B_1^3(t) = 3 \cdot (1-t)^2 \cdot t$$
$$B_2^3(t) = 3 \cdot (1-t) \cdot t^2$$
$$B_3^3(t) = t^3$$

- Para generar la curva, el VertexShader se limita a copiar la posición de los vértices de entrada. Como estamos generando una curva sobre una superficie 2D, las posiciones se definen como vec2.

```
#version 400

layout(location = 0) in vec2 VertexPosition;

void main()
{
  gl_Position = vec4( VertexPosition, 0.0, 1.0);
}
```

- El TessellationControlShader recibe como entrada una lista con los vértices de la primitiva (*gl_in*) y las variables uniformes *NumSegments* (número de segmentos para dibujar la curva) y *NumStrips* (número de líneas, que debe ser 1 en este caso)

```
#version 400

layout(vertices = 4) out;
uniform int NumSegments;
uniform int NumStrips;

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

  gl_TessLevelOuter[0] = float(NumSegments);
  gl_TessLevelOuter[1] = float(NumStrips);
}
```

- Tessellation Evaluation Shader

```
#version 400

layout(isolines) in;
uniform mat4 MVP;

void main()
{
  float u = gl_TessCoord.x;
  vec3 p0 = gl_in[0].gl_Position.xyz;
  vec3 p1 = gl_in[1].gl_Position.xyz;
  vec3 p2 = gl_in[2].gl_Position.xyz;
  vec3 p3 = gl_in[3].gl_Position.xyz;
  float u1 = (1.0-u);
  float b0 = u1*u1*u1;
  float b1 = 3*u1*u1*u;
  float b2 = 3*u1*u*u;
  float b3 = u*u*u;
  vec3 p = p0*b0+p1*b1+p2*b2+p3*b3;
  gl_Position = MVP * vec4( p, 1.0);
}
```

- El Tessellation Evaluation Shader recibe como entrada las coordenadas de cada vértice en el espacio de parámetros (*gl_TessCoord*) y los atributos de los patches (*gl_in*) y genera como salida la posición del vértice en coordenadas espaciales (*gl_Position*). Además, indica que las primitivas que provienen del TPG son de tipo *isolines*. (Otras opciones válidas son *triangles* y *quads*)
- El Fragment Shader es muy sencillo

```
#version 400

uniform vec4 LineColor;
out vec4 FragColor;

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

9.1 Los shaders de teselado

9.2 Teselado de una curva

9.3 Teselado de un cuadrilátero

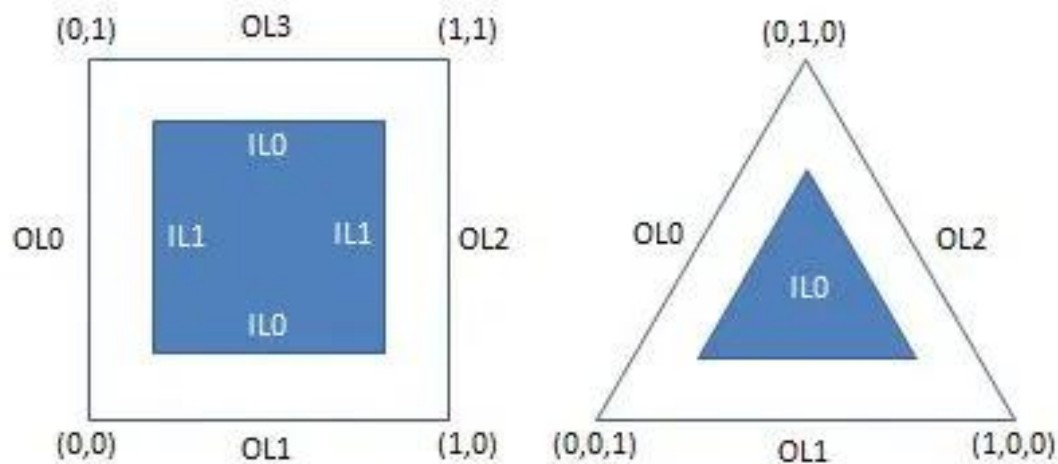
9.4 Teselado de una superficie 3D

9.5 Teselado basado en profundidad

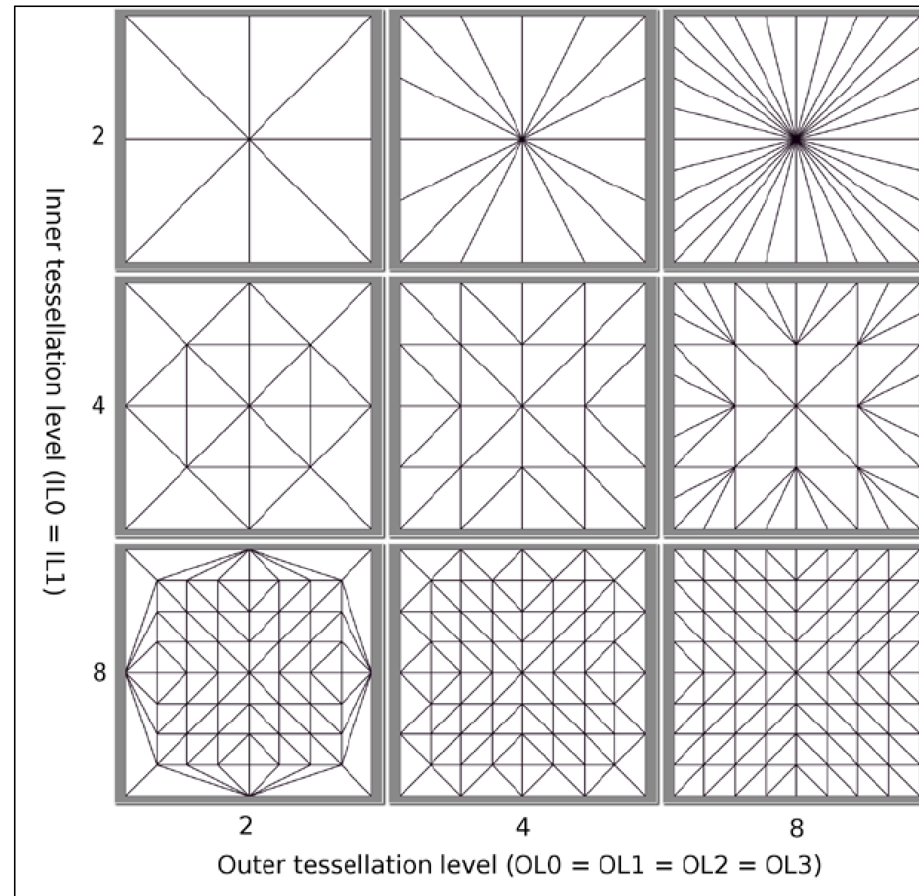
- El Tessellation Primitive Generator puede generar tres tipos de primitivas, que se indican en el TES por medio de la declaración *layout(…) in*:
  - *isolines*: el TPG genera un "*patch*" formado por un bloque rectangular de líneas paralelas. Las primitivas generadas son de tipo *line*.
  - *triangles*: el TPG genera un "*patch*" en forma triangular. Las primitivas generadas son de tipo *triangle*.
  - *quads*: el TPG genera un "*patch*" en forma de cuadrilátero. Las primitivas generadas son de tipo *triangle*.

- La forma concreta de las primitivas generadas por el TPG se configura en el TCS por medio de las salidas predefinidas *gl_TessLevelOuter[4]* y *gl_TessLevelInner[2]*.
- Los valores de estas variables tienen el siguiente significado:
  - *gl_TessLevelOuter[0]*: Número de subdivisiones de *v* para *u*=0.
  - *gl_TessLevelOuter[1]*: Número de subdivisiones de *u* para *v*=0.
  - *gl_TessLevelOuter[2]*: Número de subdivisiones de *v* para *u*=1.
  - *gl_TessLevelOuter[3]*: Número de subdivisiones de *u* para *v*=1.
  - *gl_TessLevelInner[0]*: Número de subdivisiones de *u* para valores internos de *v*.
  - *gl_TessLevelInner[1]*: Número de subdivisiones de *v* para valores internos de *u*.

- La siguiente figura muestra el espacio (*u*,*v*) para las salidas de tipo *quad* y *triangles* y el sentido de las variables *gl_TessLevelOuter* (OL) y *gl_TessLevelInner* (IL).



- Estos son algunos ejemplos de las primitivas generadas para varios valores de *gl_TessLevelOuter* y *gl_TessLevelInner*.



- Para generar las figuras anteriores se utilizarían los siguientes shaders:

  - Vertex Shader

```
#version 400

layout(location = 0) in vec2 VertexPosition;

void main()
{
  gl_Position = vec4( VertexPosition, 0.0, 1.0);
}
```

Toma como entrada una posición en formato (x,y).

– Tessellation Control Shader

```
#version 400

layout( vertices=4 ) out;
uniform int Outer;
uniform int Inner;

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

    gl_TessLevelOuter[0] = float(Outer);
    gl_TessLevelOuter[1] = float(Outer);
    gl_TessLevelOuter[2] = float(Outer);
    gl_TessLevelOuter[3] = float(Outer);
    gl_TessLevelInner[0] = float(Inner);
    gl_TessLevelInner[1] = float(Inner);
}
```

– Tessellation Evaluation Shader

```
#version 400

layout( quads, equal_spacing, ccw ) in;
uniform mat4 MVP;

void main()
{
    float u = gl_TessCoord.x;
    float v = gl_TessCoord.y;
    vec4 p0 = gl_in[0].gl_Position;
    vec4 p1 = gl_in[1].gl_Position;
    vec4 p2 = gl_in[2].gl_Position;
    vec4 p3 = gl_in[3].gl_Position;
    vec4 position = p0 * (1-u) * (1-v) +
                    p1 * u * (1-v) +
                    p3 * v * (1-u) +
                    p2 * u * v;
    gl_Position = MVP * position;
}
```

– Fragment Shader

```
#version 400

uniform vec4 LineColor;
out vec4 FragColor;

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

– En el programa OpenGL hay que activar el dibujo en modo arista e indicar que el número de vértices por primitiva "patch"son 4.

glPolygonMode(GL_FRONT, GL_LINE);

glPatchParameteri(GL_PATCH_VERTICES, 4);

9.1 Los shaders de teselado

9.2 Teselado de una curva

9.3 Teselado de un cuadrilátero

9.4 Teselado de una superficie 3D

9.5 Teselado basado en profundidad

- Una forma de modelar superficies curvas es por medio de superficies de Bezier.
- La forma general de una superficie de Bezier es de orden n es

$$P(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{n} B_i^n(u) \cdot B_j^n(v) \cdot P_{ij}$$

donde B_i^n son polinomios de Bernstein y P_{ij} son los puntos de configuración de la curva.

- Para una interpolación cúbica, las superficies de Bezier se definen por medio de 16 puntos de control y la fórmula queda

$$P(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} B_i^3(u) \cdot B_j^3(v) \cdot P_{ij}$$

- Es importante generar no solo las posiciones de los puntos sino también sus normales, para poder utilizarlas en los modelos de iluminación:

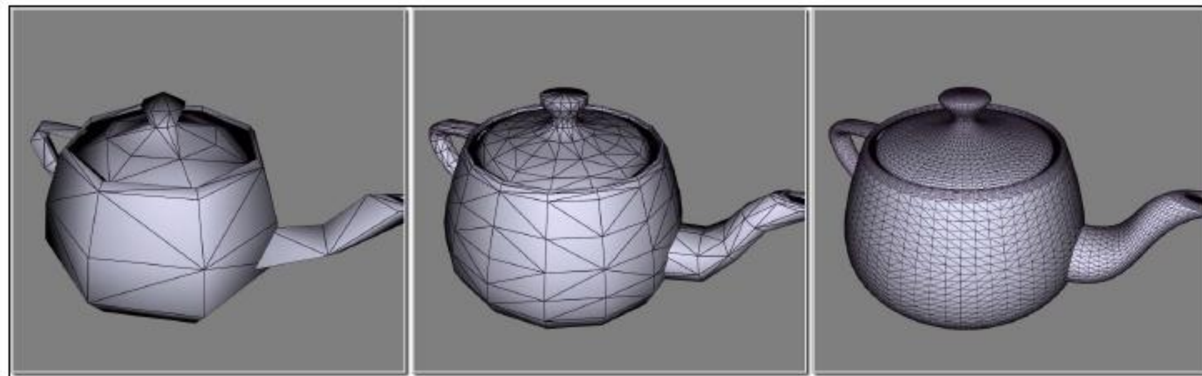
$$N(u,v) = \frac{\partial P}{\partial u} \times \frac{\partial P}{\partial v}$$

- Estas derivadas parciales se pueden calcular en términos de las derivadas de los polinomios de Bernstein

$$\frac{\partial P}{\partial u} = \sum_{i=0}^{3} \sum_{j=0}^{3} \frac{\partial B_i^3(u)}{\partial u} \cdot B_j^3(v) \cdot P_{ij}$$

$$\frac{\partial P}{\partial v} = \sum_{i=0}^{3} \sum_{j=0}^{3} B_i^3(u) \cdot \frac{\partial B_j^3(v)}{\partial v} \cdot P_{ij}$$

- Si se define un objeto como un conjunto de superfices de Bezier, descritas por medio de sus puntos de control (16 puntos por cada superficie si utilizamos interpolación cúbica), es posible dibujar el objeto con diferentes niveles de detalle utilizando los shaders de teselado.
- 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).



- Tessellation Control Shader:

```
#version 400

layout( vertices=16 ) out;
uniform int TessLevel;

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

    gl_TessLevelOuter[0] = float(TessLevel);
    gl_TessLevelOuter[1] = float(TessLevel);
    gl_TessLevelOuter[2] = float(TessLevel);
    gl_TessLevelOuter[3] = float(TessLevel);
    gl_TessLevelInner[0] = float(TessLevel);
    gl_TessLevelInner[1] = float(TessLevel);
}
```

- Tessellation Evaluation Shader:

```
#version 400

layout( quads ) in;

out vec3 Normal;
out vec4 Position;

uniform mat4 MVP;
uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;

void BernsteinPolynomials(out float[4] b, float t)
{
    float t1 = (1.0 - t);
    b[0] = t1 * t1 * t1;
    b[1] = 3.0 * t1 * t1 * t;
    b[2] = 3.0 * t1 * t * t;
    b[3] = t * t * t;
}
```

- Tessellation Evaluation Shader (sigue):

```
…
void BernsteinDeriv(out float db[4], float t)
{
    float t1 = (1.0 - t);
    db[0] = -3.0 * t1 * t1;
    db[1] = -6.0 * t * t1 + 3.0 * t1*t1;
    db[2] = -3.0 * t * t + 6.0 * t * t1;
    db[3] = 3.0 * t * t;
}

void main()
{
    // The sixteen control points
    vec4 p00 = gl_in[0].gl_Position;
    vec4 p01 = gl_in[1].gl_Position;
    vec4 p02 = gl_in[2].gl_Position;
    vec4 p03 = gl_in[3].gl_Position;
    …
```

- Tessellation Evaluation Shader (sigue):

```
…
vec4 p11 = gl_in[5].gl_Position;
vec4 p12 = gl_in[6].gl_Position;
vec4 p13 = gl_in[7].gl_Position;
vec4 p20 = gl_in[8].gl_Position;
vec4 p21 = gl_in[9].gl_Position;
vec4 p22 = gl_in[10].gl_Position;
vec4 p23 = gl_in[11].gl_Position;
vec4 p30 = gl_in[12].gl_Position;
vec4 p31 = gl_in[13].gl_Position;
vec4 p32 = gl_in[14].gl_Position;
vec4 p33 = gl_in[15].gl_Position;

float u = gl_TessCoord.x;
float v = gl_TessCoord.y;
float bu[4], bv[4];
BernsteinPolynomials(bu,u);
BernsteinPolynomials(bv,v);
…
```

- Tessellation Evaluation Shader (sigue):

```
…
float dbu[4], dbv[4];
BernsteinDeriv(dbu,u);
BernsteinDeriv(dbv,v);

Position =  p00*bu[0]*bv[0] + p01*bu[0]*bv[1]
        + p02*bu[0]*bv[2] + p03*bu[0]*bv[3]
        + p10*bu[1]*bv[0] + p11*bu[1]*bv[1]
        + p12*bu[1]*bv[2] + p13*bu[1]*bv[3]
        + p20*bu[2]*bv[0] + p21*bu[2]*bv[1]
        + p22*bu[2]*bv[2] + p23*bu[2]*bv[3]
        + p30*bu[3]*bv[0] + p31*bu[3]*bv[1]
        + p32*bu[3]*bv[2] + p33*bu[3]*bv[3];
```

- Tessellation Evaluation Shader (sigue):

```
…
vec4 du = p00*dbu[0]*bv[0] + p01*dbu[0]*bv[1]
        + p02*dbu[0]*bv[2] + p03*dbu[0]*bv[3]
        + p10*dbu[1]*bv[0] + p11*dbu[1]*bv[1]
        + p12*dbu[1]*bv[2] + p13*dbu[1]*bv[3]
        + p20*dbu[2]*bv[0] + p21*dbu[2]*bv[1]
        + p22*dbu[2]*bv[2] + p23*dbu[2]*bv[3]
        + p30*dbu[3]*bv[0] + p31*dbu[3]*bv[1]
        + p32*dbu[3]*bv[2] + p33*dbu[3]*bv[3];

vec4 dv = p00*bu[0]*dbv[0] + p01*bu[0]*dbv[1]
        + p02*bu[0]*dbv[2] + p03*bu[0]*dbv[3]
        + p10*bu[1]*dbv[0] + p11*bu[1]*dbv[1]
        + p12*bu[1]*dbv[2] + p13*bu[1]*dbv[3]
        + p20*bu[2]*dbv[0] + p21*bu[2]*dbv[1]
        + p22*bu[2]*dbv[2] + p23*bu[2]*dbv[3]
        + p30*bu[3]*dbv[0] + p31*bu[3]*dbv[1]
        + p32*bu[3]*dbv[2] + p33*bu[3]*dbv[3];
…
```

- Tessellation Evaluation Shader (fin):

```
    …
    // Normal vector
    vec3 n = normalize( cross(du.xyz, dv.xyz) );

    // Transform to clip coordinates
    gl_Position = MVP * Position;

    // Convert to camera coordinates
    Position = ModelViewMatrix * Position;
    Normal = normalize(NormalMatrix * n);
}
```

- El programa OpenGL debe asignar el número de vértices de cada primitiva “patch”:

glPatchParameteri(GL_PATCH_VERTICES, 16);

9.1 Los shaders de teselado

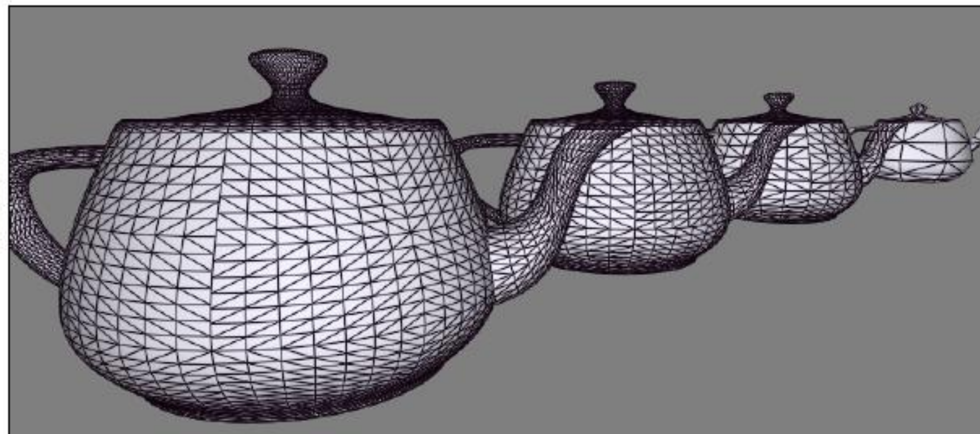
9.2 Teselado de una curva

9.3 Teselado de un cuadrilátero

9.4 Teselado de una superficie 3D

9.5 Teselado basado en profundidad

- En el ejemplo anterior el nivel de teselado era una variable uniforme que se configuraba desde el programa OpenGL. Se puede modificar el TCS para que el nivel de teselado se calcule internamente.
- Por ejemplo, se puede utilizar un nivel de teselado mayor o menor en función de la distancia a la que se encuentre el objeto a dibujar. De esta forma tendremos objetos que se pueden dibujar con diferente nivel de detalle (LoD).



- Para realizar un teselado en función de la distancia tan solo hay que modificar el TCS para elegir el valor del nivel de teselado (*TessLevel*).
- En el siguiente ejemplo se han introducido cuatro variables de configuración para calcular el nivel de teselado:
  - *MinTessLevel*: nivel de teselado mínimo.
  - *MaxTessLevel*: nivel de teselado máximo.
  - *MaxDepth*: distancia a la que el nivel alcanza el mínimo.
  - *MinDepth*: distancia a la que el nivel alcanza el máximo.
- Para calcular la distancia se necesita también la matriz ModelView.

- Tessellation Control Shader:

```
#version 400

layout( vertices=16 ) out;

uniform int MinTessLevel;
uniform int MaxTessLevel;
uniform float MaxDepth;
uniform float MinDepth;
uniform mat4 ModelViewMatrix;

void main()
{
    vec4 pos = ModelViewMatrix *
            gl_in[gl_InvocationID].gl_Position;
    float rate = (abs(p.z)-MinDepth) /
                 (MaxDepth - MinDepth);
    float depth = clamp(rate, 0.0, 1.0 );
    …
```

- Tessellation Control Shader:

```
    …

    float tessLevel = mix(MaxTessLevel, MinTessLevel,
                              depth);

    gl_TessLevelOuter[0] = float(tessLevel);
    gl_TessLevelOuter[1] = float(tessLevel);
    gl_TessLevelOuter[2] = float(tessLevel);
    gl_TessLevelOuter[3] = float(tessLevel);
    gl_TessLevelInner[0] = float(tessLevel);
    gl_TessLevelInner[1] = float(tessLevel);
    gl_out[gl_InvocationID].gl_Position =
            gl_in[gl_InvocationID].gl_Position;
}
```