Programando la GPU (III)

Shaders de geometría

flickr.com/photos/guypaterson

Bibliografía:

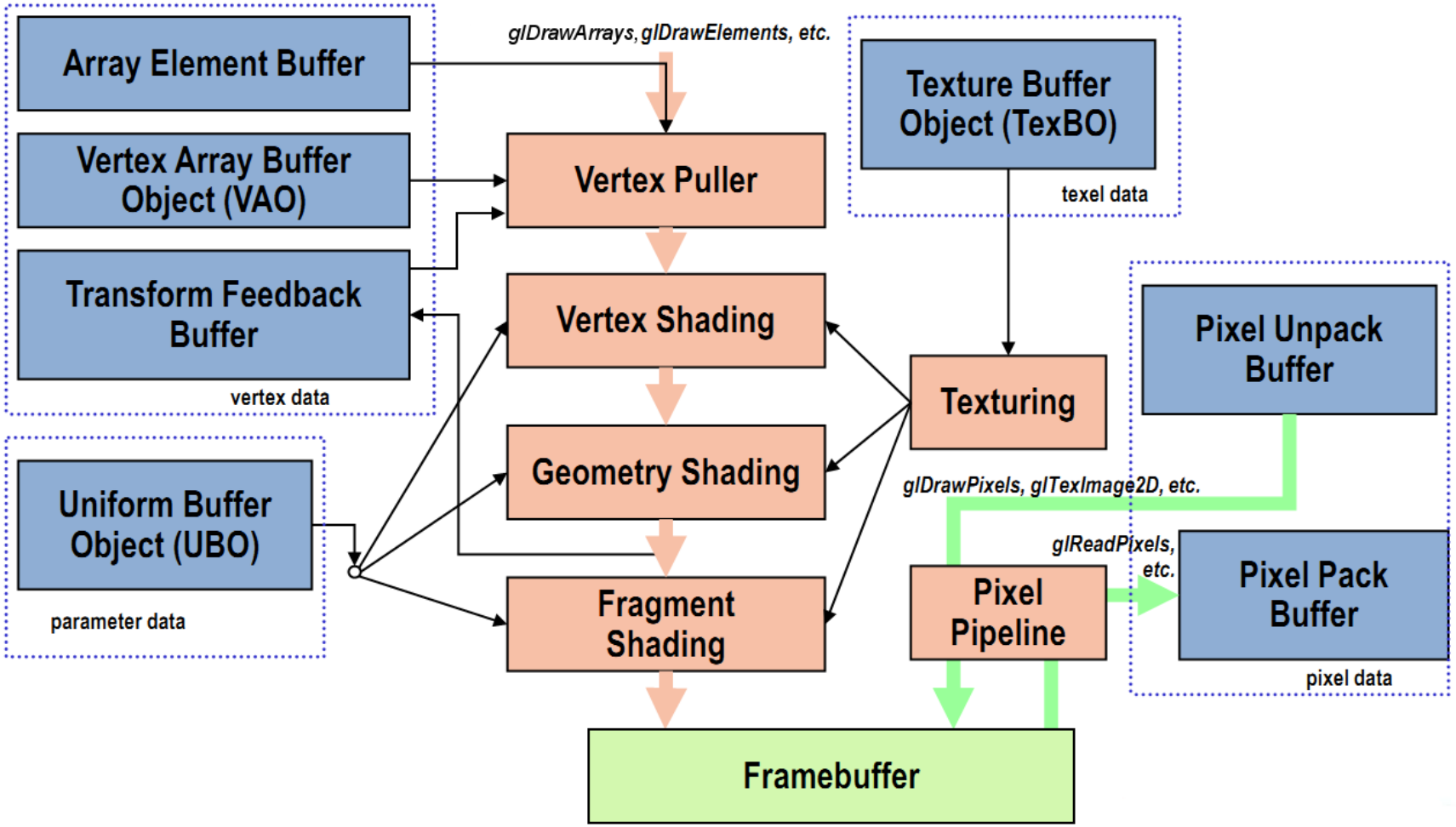
- Superbiblia, 7ª ed. pp. 333-364, 397-403

Índice

- Introducción
- El shader de geometría
  - Tipos de primitiva de entrada/salida
  - Paso de información entre shaders
- ¿Qué se puede hacer en un shader de geometría?
  - Descartar polígonos
  - Modificar la geometría
  - Aumentar la geometría
  - Cambiar el tipo de primitiva
- Nuevas primitivas

Introducción

La tubería de OpenGL 3.3



Introducción

- El shader de geometría se introdujo inicialmente como una extensión, pero a partir de OpenGL 3.2 está en el *core*
- Procesa primitivas completas (en vez de vértices o fragmentos aislados)
  - Tiene acceso a todos los vértices de un triángulo o línea
- Recuerda:
  - Shader de vértice: entra un vértice, sale un vértice. Sin acceso a otros vértices
  - Shader de fragmento: puede descartar el fragmento, y sólo procesa uno cada vez
- Puede: cambiar el tipo de primitiva, introducir nuevas primitivas o destruir la primitiva actual

Introducción

- Cuando en el programa actual hay un shader de geometría:
  - éste recibe la salida de varias invocaciones del shader de vértice y
  - genera las primitivas que, una vez rasterizadas, serán la entrada del shader de fragmento
- Usos típicos: generar siluetas, transformar primitivas completas, mostrar normales, implementar nivel de detalle limitado, dibujar varias veces una primitiva, etc.

Introducción

Ejemplo: shader de geometría identidad

```
// Shader de la página 334 de la Superbiblia 7ª ed.

#version 430

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

void main() {
  int i;

  for (i = 0; i < gl_in.length(); i++) {
    gl_Position = gl_in[i].gl_Position;
    EmitVertex();
  }
  EndPrimitive();
}
```

Introducción

Ejemplo: shader de geometría mínimo

- Un shader de geometría debe declarar el tipo de primitiva que recibe, el tipo de primitiva que genera y el número máximo de vértices que podrá generar
  - El shader de geometría anterior recibe un triángulo y genera una cinta de triángulos (aunque como máximo, con 3 vértices)
  - Únicamente propaga la posición (el resto de atributos no son accesibles en el shader de fragmento)

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

El shader de geometría

Tipos de primitivas de entrada y salida

- El tipo de primitiva de entrada se define en el shader con la instrucción:
  - `layout (<tipo>) in;`

  donde `<tipo>` puede ser:
    - `points`, `lines`, `triangles`
    - `lines_adjacency`, `triangles_adjacency`
- Debe coincidir con el usado al dibujar las primitivas:
  - `points`: `GL_POINTS`
  - `lines`: `GL_LINES`, `GL_LINE_LOOP`, `GL_LINE_STRIP`
  - `triangles`: `GL_TRIANGLES`, `GL_TRIANGLE_FAN`, `GL_TRIANGLE_STRIP`
  - `lines_adjacency`: `GL_LINES_ADJACENCY`, `GL_LINE_STRIP_ADJACENCY`
  - `triangles_adjacency`: `GL_TRIANGLES_ADJACENCY`, `GL_TRIANGLE_STRIP_ADJACENCY`

El shader de geometría

Tipos de primitivas de entrada y salida

- El tipo de primitiva de salida y el número máximo de vértices se define mediante:
  - `layout (<tipo>, max_vertices = <n>) out;`
  - donde `<tipo>` puede ser:
    - `points`, `line_strip`, `triangle_strip`
  - y `<n>` es el número máximo de vértices que se pueden generar.
  - El número máximo de vértices que puede generar un shader de geometría se puede consultar con: `glGetIntegerv(GL_MAX_GEOMETRY_OUTPUT_VERTICES, &i)` (mínimo 256, RTX 3070: 1024)

El shader de geometría

Paso de información entre shaders

- Los shader de geometría tienen las siguientes variables predefinidas

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

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

En estas variables escribe el shader de vértice

El tamaño de este array (es decir, el número de vértices de la primitiva de entrada) se puede obtener con **`gl_in.length()`**

¡No es un array!

El shader de geometría

Paso de información entre shaders

- El número de vértices de entrada depende del tipo de primitiva que se está dibujando:
  - `points` (1), `lines` (2), `triangles` (3), `lines_adjacency` (4), `triangles_adjacency` (6)
- Procesamiento de la primitiva de entrada:

```
for (i = 0; i < gl_in.length(); i++) {
   gl_Position = gl_in[i].gl_Position;
   EmitVertex();  // generar un nuevo vértice
 }
 EndPrimitive(); // generar una nueva primitiva
```

El shader de geometría

Paso de información entre shaders

- Un shader de geometría puede generar varias primitivas independientes (con varias llamadas a `EndPrimitive()`)
- Si un shader de geometría acaba sin una llamada a `EndPrimitive()`, GLSL la llama automáticamente

El shader de geometría

Paso de información entre shaders

- ¿Y cómo se propagan atributos definidos por el usuario?

Shader de vértice

```
#version 330

in vec4 pos;
in vec3 normal;

out vec4 color;
out vec3 N;
…
```

Shader de geometría

```
#version 330

layout (…) in;
layout (…) out;

in vec4 color[];
in vec3 N[];

out vec4 colorF;
…
```

Shader de fragmento

```
#version 330

in vec4 colorF;
out vec4 finalColor;

…
```

Tienes un ejemplo completo en ej61

El shader de geometría

Paso de información entre shaders

- Otra opción: con bloques de interfaz

Shader de vértice

```
#version 330

in vec4 pos;
in vec3 normal;

out VertexData {
  vec4 color;
  vec3 N;
} vertex;
…
vertex.N = …
```

Shader de geometría

```
#version 330

layout (…) in;
layout (…) out;

in VertexData {
  vec4 color;
  vec3 N;
} vtcs[];

out vec4 colorF;
…
colorF = vtcs[i].color;
```

Shader de fragmento

```
#version 330

in vec4 colorF;
out vec4 finalColor;

…
```

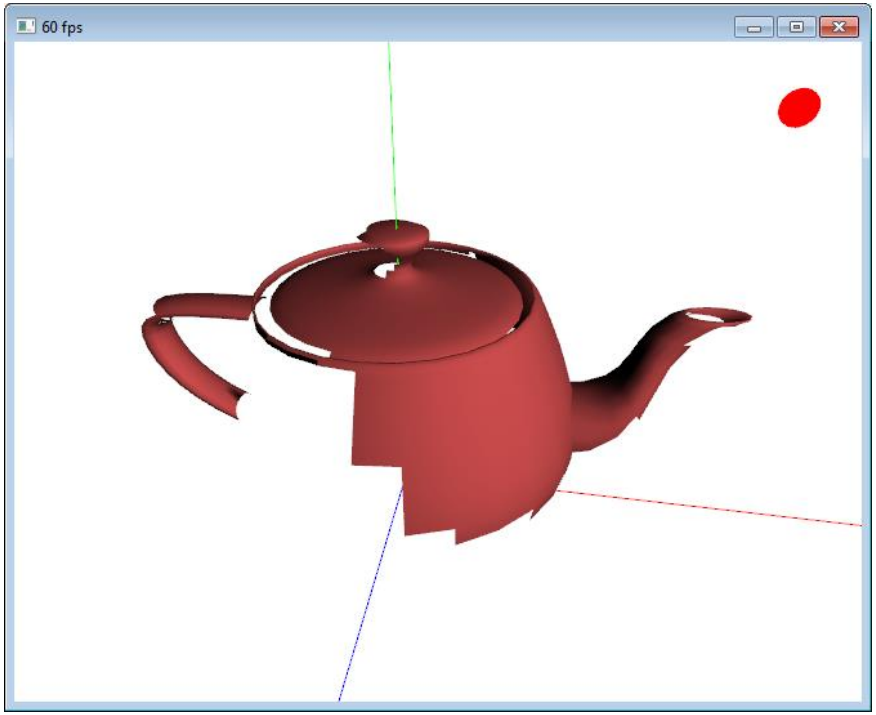
El shader de geometría

Paso de información entre shaders

- En cada llamada a `EmitVertex()`, GLSL adjuntará el valor actual de todas las variables de tipo `out` del shader
  - ¡Cuidado! El contenido de las variables de tipo **`out`** se invalida en cada llamada (las tienes que rellenar siempre, aunque no cambien)
- Si en un shader de geometría no se genera ninguna primitiva completa, no llegará nada al rasterizador

Descartando polígonos

- Descartar aquellos polígonos que den la espalda a la esfera



Descartando polígonos

ej6-2.vert

```
#version 420

$GLMatrices

in vec4 position;
in vec3 normal;

// Posición de la fuente en el espacio de la cámara
uniform vec3 lightpos;
// Color difuso del objeto
uniform vec4 diffuseColor = vec4(0.8, 0.3, 0.3, 1.0);

out vec4 color;
void main() {
  // Normal del vértice en el espacio de la cámara
  vec3 eN = normalize(normalMatrix *normal);
  // Vértice en el espacio de la cámara
  vec4 eposition=modelviewMatrix * position;
  // Cálculo de la iluminación
  color = max(0.0, dot(eN, normalize(lightpos-eposition.xyz))) * diffuseColor;
  gl_Position = eposition;
}
```

Descartando polígonos

ej6-2.geom

```
#version 420
$GLMatrices

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

// Posición de la esfera en el espacio de la cámara
uniform vec4 sphere;

in vec4 color[3];
out vec4 fragColor;

void main() {
  // Calculando la normal en el espacio de la cámara
  vec3 ab = gl_in[1].gl_Position.xyz - gl_in[0].gl_Position.xyz;
  vec3 ac = gl_in[2].gl_Position.xyz - gl_in[0].gl_Position.xyz;
  vec3 normal = cross(ab, ac);
  // Vector desde el primer vértice a la esfera (espacio de la cámara)
  vec3 vt = vec3(sphere-gl_in[0].gl_Position);
  if (dot(vt, normal) > 0.0) {
    for (int i = 0; i < gl_in.length(); i++) {
      gl_Position = projMatrix * gl_in[i].gl_Position;
      fragColor = color[i];
      EmitVertex();
    }
    EndPrimitive();
  }
}
```

Descartando polígonos

ej6-2.frag

```
#version 330

in vec4 fragColor;

out vec4 finalColor;


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

Modificando la geometría

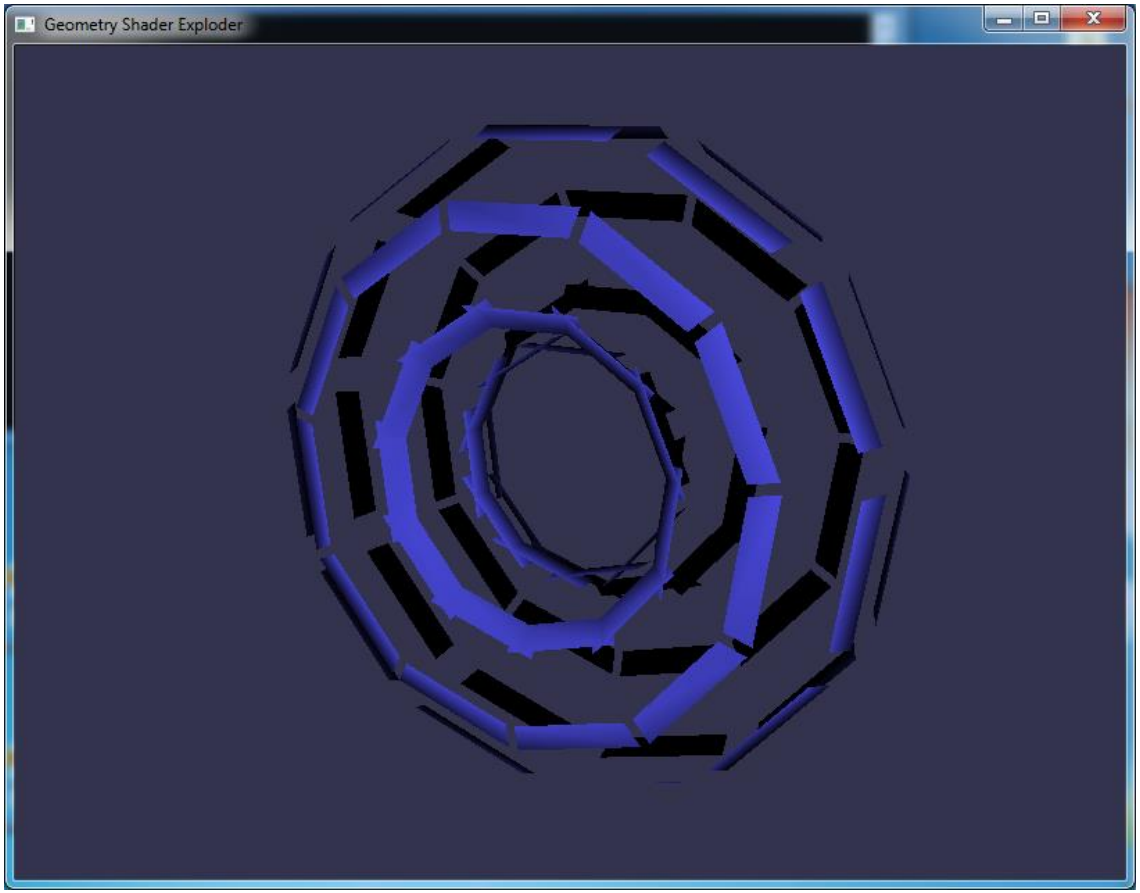
- En el shader de geometría tenemos acceso a los triángulos individuales (incluso si el modelo se definió usando cintas o abanicos)
- Esto nos permite tratar cada triángulo independientemente de los demás, como por ejemplo, para “explotar” un modelo

```
#version 330
layout (triangles) in;
layout (triangle_strip, max_vertices = 3) out;
in Vertex
{
    vec3 normal;
    vec4 color;
} vertex[];
out vec4 color;
uniform mat4 mvpMatrix;
uniform float push_out;

void main(void)
{
  vec3 face_normal = normalize(cross(
    gl_in[1].gl_Position.xyz - gl_in[0].gl_Position.xyz,
    gl_in[2].gl_Position.xyz - gl_in[0].gl_Position.xyz));
  for (int n = 0; n < gl_in.length(); n++) {
    color = vertex[n].color;
    gl_Position = mvpMatrix * vec4(gl_in[n].gl_Position.xyz +
              face_normal * push_out, gl_in[n].gl_Position.w);
    EmitVertex();
  }
  EndPrimitive();
}
```

Página 321
Superbiblia 6ª ed.

Modificando la geometría

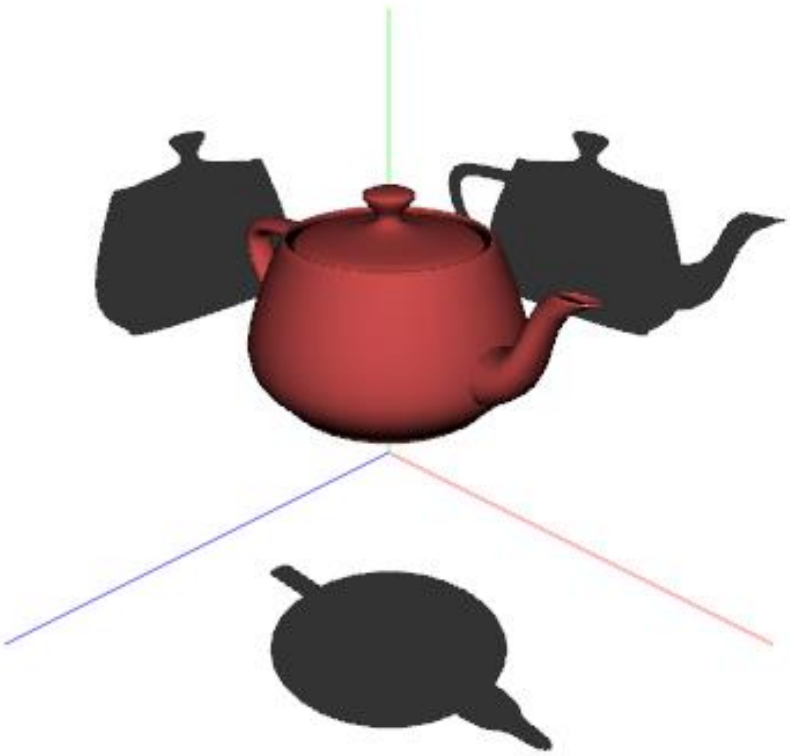


Aumentando la geometría

- Aumentar la geometría implica crear nuevas primitivas en el shader
- Se puede usar, por ejemplo, para copiar triángulos (p.e., para generar reflejos en el shader de geometría), hacer una teselación limitada (hasta 2x, 4x)…

Aumentando la geometría

ej6-3



Aumentando la geometría

ej6-3.vert

```
#version 420

$GLMatrices

in vec4 position;
in vec3 normal;

// Posición de la fuente en el espacio de la cámara
uniform vec3 lightpos;
// Color difuso del objeto
uniform vec4 diffuseColor = vec4(0.8, 0.3, 0.3, 1.0);

out vec4 color;

void main() {
  // Normal del vértice en el espacio de la cámara
  vec3 eN = normalize(normalMatrix *normal);
  // Vértice en el espacio de la cámara
  vec4 eposition=modelviewMatrix * position;
  // Cálculo de la iluminación
  color = max(0.0, dot(eN, normalize(lightpos-eposition.xyz))) * diffuseColor;
  gl_Position = modelMatrix * position;
}
```

```
#version 330
$GLMatrices
layout (triangles) in;
layout (triangle_strip, max_vertices = 12) out;
uniform vec4 projColor = vec4(0.2, 0.2, 0.2, 1.0); // color de las proyecciones
in vec4 color[3];
out vec4 fragColor;
void main() {
  mat4 projviewMatrix = projMatrix * viewMatrix;
  // Geometría "normal"
  for (int i = 0; i < gl_in.length(); i++) {
    gl_Position = projviewMatrix * gl_in[i].gl_Position;
    fragColor = color[i];
    EmitVertex();
  }
  EndPrimitive();
  // "Planta"
  for (int i = 0; i < gl_in.length(); i++) {
    gl_Position = projviewMatrix*(gl_in[i].gl_Position*vec4(1.0,0.0,1.0,1.0));
    fragColor = projColor;
    EmitVertex();
  }
  EndPrimitive();
}
```

ej6-3.geom

```
  // Perfil
  for (int i = 0; i < gl_in.length(); i++) {
    gl_Position =  projviewMatrix *
       (gl_in[i].gl_Position*vec4(0.0,1.0,1.0,1.0));
    fragColor = projColor;
    EmitVertex();
  }
  EndPrimitive();

  // Alzado
  for (int i = 0; i < gl_in.length(); i++) {
    gl_Position =  projviewMatrix *
          (gl_in[i].gl_Position*vec4(1.0,1.0,0.0,1.0));
    fragColor = projColor;
    EmitVertex();
  }
  EndPrimitive();
}
```

ej6-3.geom (2)

Aumentando la geometría

- Podemos pedir a OpenGL que ejecute el shader de geometría más de una vez por cada primitiva de entrada:
  - `layout (invocations = <n>, <tipo>) in;`

  donde:
    - `<tipo>`: `points`, `lines`, `triangles`, `lines_adjacency`, `triangles_adjacency`
    - `<n>`: número de ejecuciones del shader por cada primitiva
- Dentro del shader podemos distinguir el número de invocación que se está procesando con la variable:
  - `gl_InvocationID`: 0..<n-1>

Aumentando la geometría

- Ejemplo anterior con 4 invocaciones:

ej6-4

```
#version 330
$GLMatrices
layout (triangles, invocations = 4) in;
layout (triangle_strip, max_vertices = 12) out;

uniform vec4 projColor = vec4(0.2, 0.2, 0.2, 1.0); // color de las proyecciones
in vec4 color[3];
out vec4 fragColor;

void main() {
  mat4 projviewMatrix = projMatrix * viewMatrix;
  for (int i = 0; i < gl_in.length(); i++) {
    vec4 proj = vec4(1.0);
    if (gl_InvocationID < 3) {
      proj[gl_InvocationID] = 0.0;
      fragColor = projColor;
    } else
      fragColor = color[i];
    gl_Position = projviewMatrix * (gl_in[i].gl_Position * proj);
    EmitVertex();
  }
  EndPrimitive();
}
```

Cambiando el tipo de primitiva

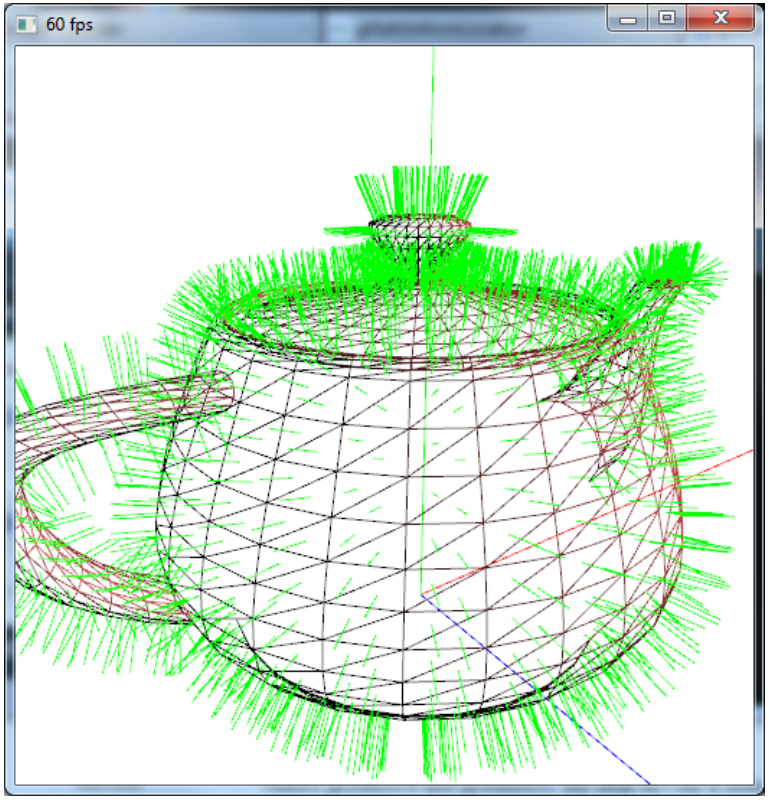
- El shader de geometría declara tanto el tipo de primitiva de entrada como el de salida, y se pueden usar tipos distintos:

```
layout (triangles) in;
layout (line_strip, max_vertices = 4) out;
```

- Un shader de geometría no puede generar dos tipos distintos de geometría

Cambiando el tipo de primitiva

ej6-5



Cambiando el tipo de primitiva

ej6-5.vert

```
#version 330

$GLMatrices

in vec4 position;
in vec3 normal;

out vec3 ecnormal;

void main() {
  // Normal del vértice en el espacio de la cámara
  ecnormal = normalize(normalMatrix *normal);
  // Vértice en el espacio de la cámara
  gl_Position = modelviewMatrix * position;
}
```

ej6-5.frag

```
#version 330

in vec4 fragColor;

out vec4 finalColor;


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

Cambiando el tipo de primitiva

ej6-5.geom (1)

```
#version 330

$GLMatrices

layout (triangles) in;
layout (line_strip, max_vertices = 8) out;

// Color de la normal de vértice
uniform vec4 vertNColor;
// Color de la normal de cara
uniform vec4 faceNColor;

uniform float normalLength;
uniform bool showFaceNormal;
uniform bool showVertexNormal;

in vec3 ecnormal[];
out vec4 fragColor;
```

Cambiando el tipo de primitiva

ej6-5.geom (2)

```
void main() {
  vec3 ab = gl_in[1].gl_Position.xyz - gl_in[0].gl_Position.xyz;
  vec3 ac = gl_in[2].gl_Position.xyz - gl_in[0].gl_Position.xyz;
  vec3 fnormal = normalize(cross(ab, ac));
  vec3 center = (gl_in[0].gl_Position.xyz + gl_in[1].gl_Position.xyz +
                          gl_in[2].gl_Position.xyz)/3;
  if (dot(fnormal, center) < 0) {
    if (showFaceNormal) {
      gl_Position = projMatrix * vec4(center, 1.0);
      fragColor = faceNColor;
      EmitVertex();
      gl_Position = projMatrix *
                          (vec4(center + fnormal*normalLength, 1.0));
      fragColor = faceNColor;
      EmitVertex();
      EndPrimitive();
    }
```

Cambiando el tipo de primitiva

ej6-5.geom (y 3)

```
    if (showVertexNormal) {
      for (int i = 0; i < gl_in.length(); i++) {
        gl_Position = projMatrix * gl_in[i].gl_Position;
        fragColor = vertNColor;
        EmitVertex();
        gl_Position = projMatrix * (gl_in[i].gl_Position +
                      vec4(ecnormal[i] * normalLength, 0.0));
        fragColor = vertNColor;
        EmitVertex();
        EndPrimitive();
      }
    } // if (showVertexNormal)
  } // if (dot(fnormal, center) < 0)
} // main
```

Nuevas primitivas

- Los shaders de geometría introdujeron cuatro nuevos tipos de primitivas, que incorporan información de adyacencia:
  - `GL_LINES_ADJACENCY`
  - `GL_LINE_STRIP_ADJACENCY`
  - `GL_TRIANGLES_ADJACENCY`
  - `GL_TRIANGLE_STRIP_ADJACENCY`
- Permiten pasar información sobre primitivas adyacentes

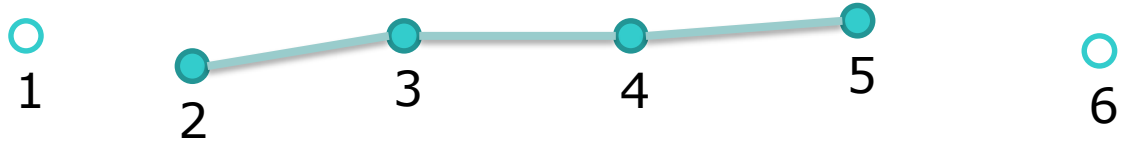
Nuevas primitivas

- `GL_LINES_ADJACENCY`: define un segmento de línea, y sus dos vértices adyacentes:

  - Si no hay un shader de geometría instalado, los vértices adyacentes se ignoran (por cada 4 vértices consecutivos, se dibuja un segmento entre los vértices 2 y 3)

Nuevas primitivas

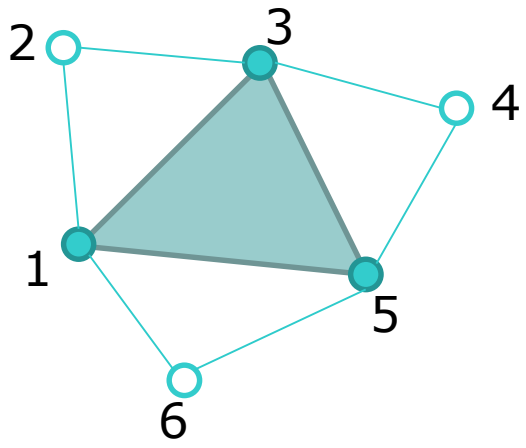
- GL_LINE_STRIP_ADJACENCY



  - El shader de geometría recibirá las líneas con adyacencia: 1234, 2345 y 3456
  - Si no hay shader de geometría, define una polilínea que conecta los vértices desde el segundo hasta el penúltimo (N segmentos = N+3 vértices)

Nuevas primitivas

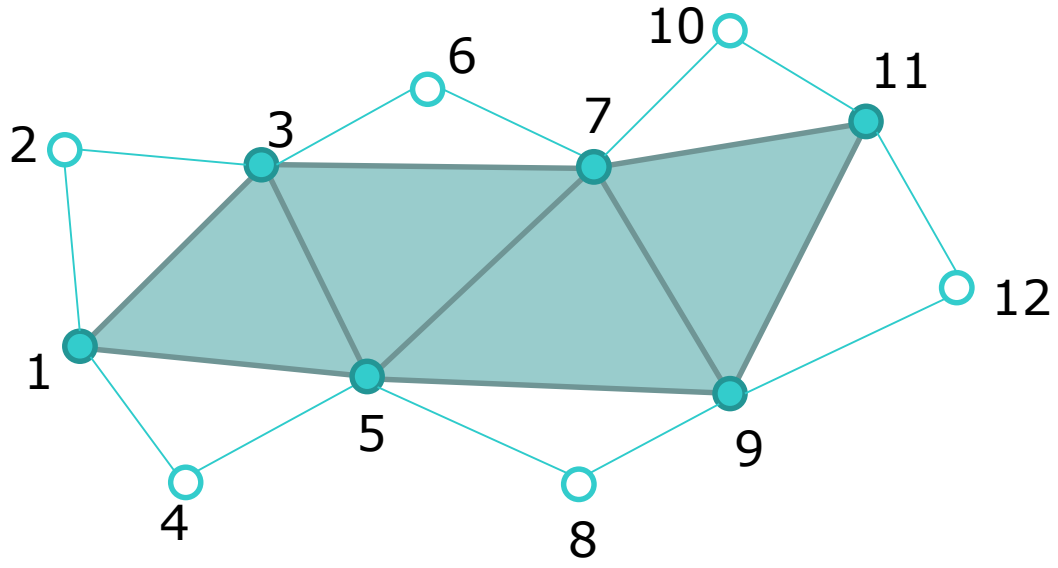
- GL_TRIANGLES_ADJACENCY



  - Si no hay shader de geometría, cada 6 vértices definen un triángulo con los vértices 1, 3 y 5 (N triángulos = 6N vértices)

Nuevas primitivas

- `GL_TRIANGLE_STRIP_ADJACENCY`



  - Define una cinta de triángulos a partir de los vértices impares (N triángulos ~ 2N vértices)

Ver apartado 10.1.14 de la especificación OpenGL 4.6

Nuevas primitivas

- Si hay un shader de geometría instalado, este puede procesar las primitivas con adyacencia como desee
- Por ejemplo, puede usar una primitiva `GL_LINES_ADJACENCY` para definir una primitiva de 4 vértices