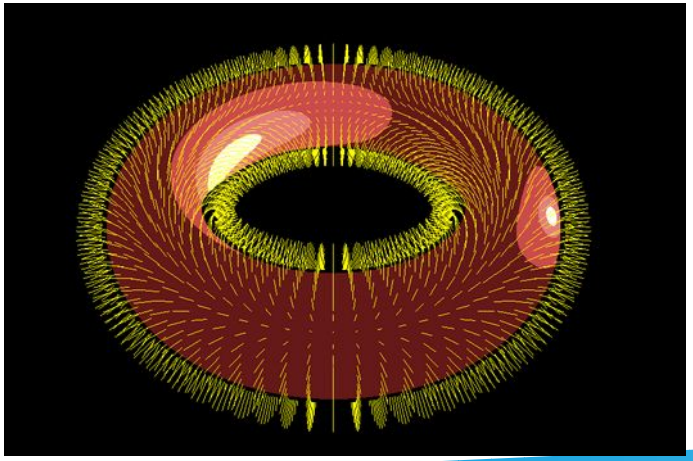
https://learnopengl.com/Advanced-OpenGL/Geometry-Shader

Advanced GLSL

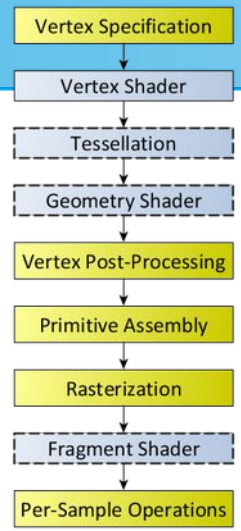
Geometry shader



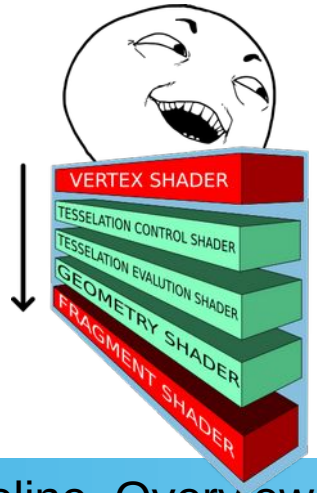
Geometry shader

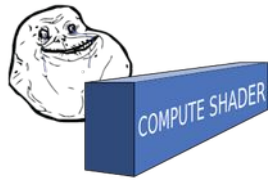
- Fonction
- Principe
- Aspects techniques

Fonction / intérêt



- Vous connaissez déjà le vertex et le fragment shader
  - Les deux sont indispensables pour faire un rendu
  - Mais il en existe d'autres comme **le geometry shader**
- Il fait partir du pipeline « géométrique » du pipeline Graphique
  - Il est optionnel
  - Il intervient entre le vertex et le fragment shader
- Les autres shader
  - Tessellation Control Shader , Tessellation Evaluation Shader
  - Mesh shader
  - Compute shader (indépendant de la géométrie et rendu)





https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

Geometry shader : Fonction / intérêt

- Permet de convertir des primitives graphiques via le GPU
  - A partir de primitive d'entrée
    - Au sens élément opengl : points, segments, triangles
    - Créer d'autres primitives : points, segments, triangles
  - On peut avoir plus de primitives et de sommets en sortie qu'en entrée
- Exemple : particules
  - Convertir des points en ensemble de triangles,
    - Un point = position de la particule

      => convertir en un ensemble de triangle = une sphère

Geometry Shader

- Déclaration du type de primitive en entrée à traiter (5 possibilités):
  - **points**: pour les primitives GL_POINTS,
  - **lines**: pour GL_LINES ou GL_LINE_STRIP,
  - **lines_adjacency**: pour GL_LINES_ADJACENCY ou GL_LINE_STRIP_ADJACENCY ,
  - **triangles**: GL_TRIANGLES, GL_TRIANGLE_STRIP ou GL_TRIANGLE_FAN
  - **triangles_adjacency** : GL_TRIANGLES_ADJACENCY ou GL_TRIANGLE_STRIP_ADJACENCY
- Directement dans le shader avec la directive :

```
layout(points) in; // on traite des points en entrée
```

Geometry Shader

- Déclaration du type de primitive en sortie
- Uniquement 3 possibilités :
  - points
  - line_strip
  - triangle_strip
- **On spécifie également le nombre max de sommets en sortie**
  - Ce nb max de sommets vaut par invocation du geometry shader
- Directement dans le shader avec la directive :

```
layout(line_strip, max_vertices = 2) out; // et on veut des line_strip en sortie (2 sommets)
```

2 mots sur les built-in variables de GLSL

- built-in variable = variable intégrée (~variable prédéfinie / globale)
- Exemple :
  - gl_Position = vecteur de sortie du vertex shader
  - gl_PointSize = taille d'un point en pixel
    - marche si glEnable(GL_PROGRAM_POINT_SIZE); // cmd opengl
  - ...
  - gl_FragCoord = coordonnées du fragment dans la fenêtre
  - gl_FrontFacing = booléen = vrai si face avant faux sinon

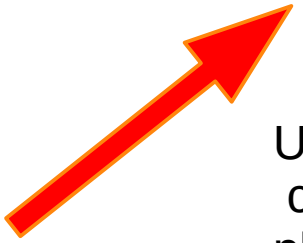
2 mots sur les built-in variable de GLSL

- Il existe une variable intégrée nommé « gl_in »

Le vertex shader à la sortie prédéfinie

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

Voir bloc d'interface
plus loin



Le geometry shader à l'entrée prédéfinie

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

Un tableau de structures,
car pour une primitive on peut avoir
plusieurs sommets (triangle, segment)

Geometry Shader : création des primitives

- À l'aide de 2 instructions :
  - **EmitVertex();**
    - crée un « vertex » à partir des valeurs de « gl_Position » spécifiées
  - **EndPrimitive();**
    - Crée une primitive à partir des vertices définis précédemment

```
#version 330 core
layout (points) in;
layout (line_strip, max_vertices = 2) out;

void main() {
    gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0,
0.0);
    EmitVertex();

    gl_Position = gl_in[0].gl_Position + vec4( 0.1, 0.0, 0.0,
0.0);
    EmitVertex();

    EndPrimitive();
}
```

Que fait ce code ?

Geometry Shader : création des primitives

- On peut créer plusieurs primitives de sortie pour 1 primitive d'entrée

```
#version 330 core
layout (points) in;
layout (line_strip, max_vertices = 2) out;

void main() {
    gl_Position = … ;      EmitVertex();
    gl_Position = … ;      EmitVertex();
    EndPrimitive();
    gl_Position = … ;      EmitVertex();
    gl_Position = … ;      EmitVertex();
    EndPrimitive();
    gl_Position = … ;      EmitVertex();
    gl_Position = … ;      EmitVertex();
    EndPrimitive();
}
```

Geometry Shader :

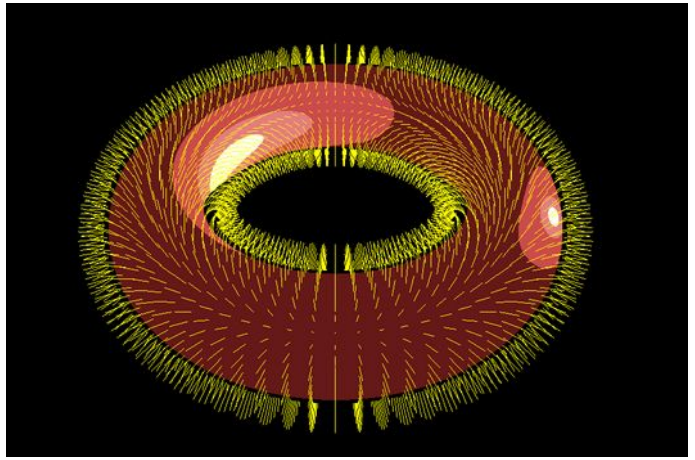
- Transmission de données vers le fragment shader
- Rappel : on travaille sur des primitives = ensemble de points
- En sortie on déclare
  - une valeur (ou structure/bloc, cf diapo plus loin)
  - une valeur par vertex émis

```
// geometry shader
in vec3 vertexColor[]; // envoyé via vertex shader
out vec3 fColor; // couleur en sortie par vertex pour le fragment shader

void main() {
    gl_Position = … ; fColor =vertexColor[0] ;    EmitVertex();
    gl_Position = … ; fColor =vec3(1.,1.,1.) ;     EmitVertex();
    EndPrimitive();
    gl_Position = … ; fColor =vertexColor[1] ;   EmitVertex();
    gl_Position = … ; fColor =vec3(1.,1.,1.) ;    EmitVertex();
    EndPrimitive();
...
```

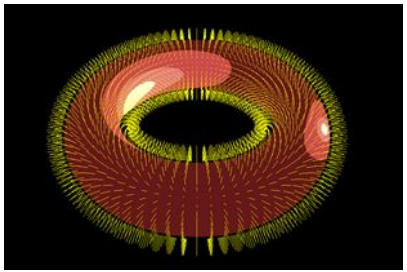
Exemple : affichage des normales

- Préparation des données (comme d'hab)
  - VBO (pos + normale) + var uniform MVP,…
- **Préparer le geometry shader** (idem vertex et fragment shader)
  - Le créer,
  - le charger,
  - le compiler,
  - l'attacher au programme shader



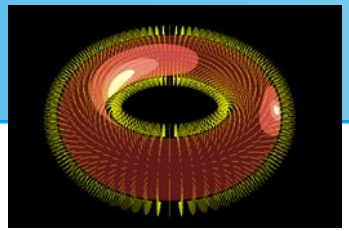
```
// seul différentce à la création → const GL_GEOMETRY_SHADER
GLuint GeometryShaderID = glCreateShader(GL_GEOMETRY_SHADER);
// le reste c'est pareil
glShaderSource(GeometryShaderID, 1, &GeometrySourcePointer , NULL);
glCompileShader(GeometryShaderID);
glAttachShader(ProgramID, GeometryShaderID);
```

Exemple : affichage des normales



- On fait un 1er rendu uniquement des normales
  - On active le shader des normales
    - glUseProgram(programIDNormal);
  - En openGL on demande à afficher que les sommets
    - glDrawElements(GL_POINTS, …
  - Les sommets sont convertis en segment dans le geometry shader

Exemple : affichage des normales



- Le vertex shader : rien ne change

```
gl_Position = VIEW * MODEL * vec4(position,1);
mat3 normalMatrix = mat3(transpose(inverse( VIEW * MODEL)));
 geomNormal = normalize(normalMatrix * normale)); // en out
```

- Le geometry shader :

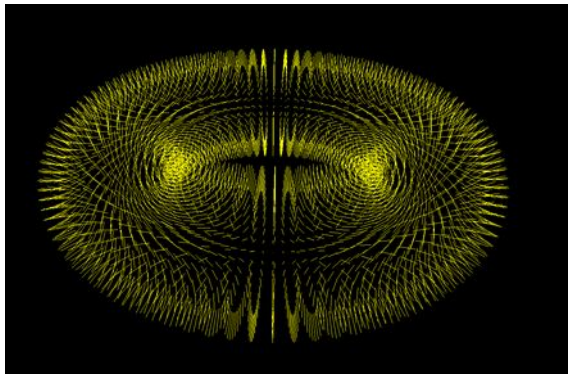
```
layout(points) in; // on traite des points en entrée
layout(line_strip, max_vertices = 2) out; // et on veut des line_strip en sortie (2 sommets)
….
  gl_Position =  PERSPECTIVE * gl_in[0].gl_Position ;
  EmitVertex(); // création d'un sommet
  gl_Position = PERSPECTIVE *  (gl_in[0].gl_Position + .1* vec4(geomNormal,0));
  EmitVertex(); // création d'un sommet
EndPrimitive(); // création d'une primitive (ici inutile car une seule primitive)
```

Exemple : affichage des normales

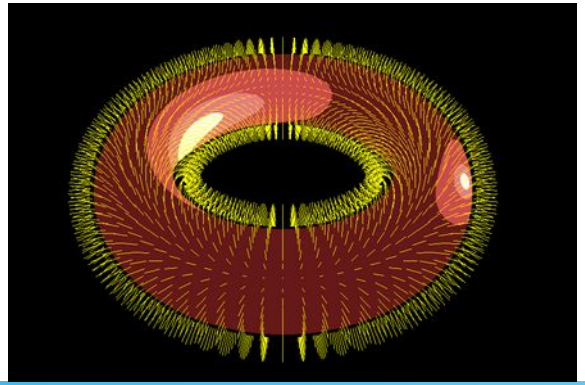
- Le fragment shader : trop facile

```
finalColor= vec4(1.,1.,0,1); // var en out
```

Et voilà le résultat

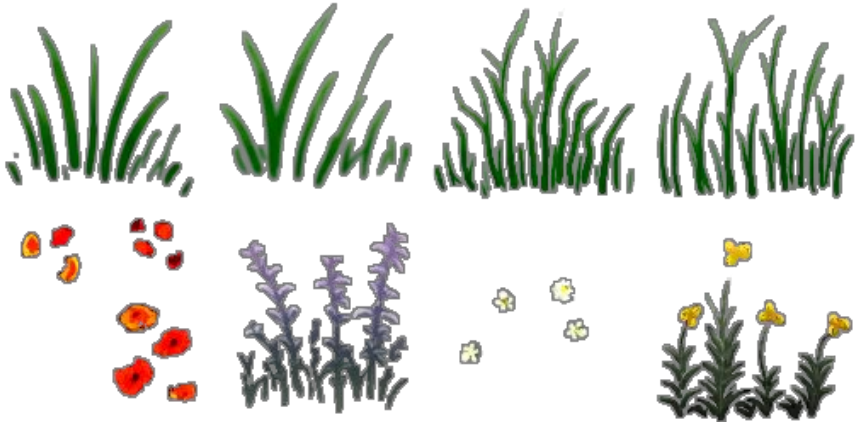


ATTENTION pour avoir celui-ci
il faut faire une deuxième passe de rendu normal



Exemple à faire : champ de fleurs

- Données
  - Une bitmap contenant des dessins de plantes



Bonne pratique : Blocs d'interface

- Spécification GLSL
  - Structuration de la transmission des données multiples entre shader
  - Permet de gérer toutes les variables de sortie dans une seule structure (=block)
- On définit une structure précédée de **Out** (ou **In**)

Nom de la structure block

```
out VS_OUT
{
    vec2 TexCoords;
    Vec3 Color ;
} vs_out;
```

Nom de variable

Et on l'utilise comme une structure

```
void main()
{
  gl_Position = projection * view * model * vec4(aPos, 1.0);
    vs_out.TexCoords = aTexCoords;
    vs_out.color = acolor ;

}
```

Bonne pratique : Blocs d'interface

- Dans le shader suivant
  - On déclare la même structure (même nom de structure)
  - Mais on peut donner un nom de variable différent

```
in VS_OUT
{
    vec2 TexCoords;
    Vec3 Color ;
} fs_in;
```

Nom de variable

Pour **ajouter une variable** out/in entre les shaders
Il suffit **d'ajouter un champ à la structure**

```
void main()
{
    FragColor = mix(texture(texture, fs_in.TexCoords), fs_in.color);
}
```