OpenGL 4 - Pour aller plus loin...

- POGLA/IG3DA -

Jonathan Fabrizio

http://jo.fabrizio.free.fr

http://www.lrde.epita.fr/~jonathan

LRDE - EPITA

Version : Mon Nov 23 10:08:26 2020

OpenGL : Pour aller plus loin

- *Compute Shader*
- *Geometry Shader*
- *Tessellation Shader*

Compute shader

- Shader en dehors du *pipeline graphic*, en dehors de la chaine de rendu
- Permet :
  - de faire du calcul,
  - de mettre à jour une animation entre deux rendus,
  - des traitements/des effets sur l'image ou dans le cas du MRT...

Compute shader : avantages/inconvénients

Avantages

- Parallélisation
- Pendant que le GPU travaille, le CPU peut faire autre chose...
- Pour une animation, le *compute shader* peut mettre à jour les données sans migrer les données de la *VRAM*

Inconvénients vis à vis de la programmation CPU

- Plus compliqué/bas niveau
- Parallélisation contrainte

Avantage vis à vis d'autres moyens de programmer le GPU

- Opengl (GLSL) donc Portable (pour combien de temps encore ?)
- Peu de chose en plus par rapport aux shaders classiques

Compute shader

Plusieurs threads qui font exactement la même chose !

```
...
if (...) {
  ...
  a=a*c; ⊗
  ...
}
...
```

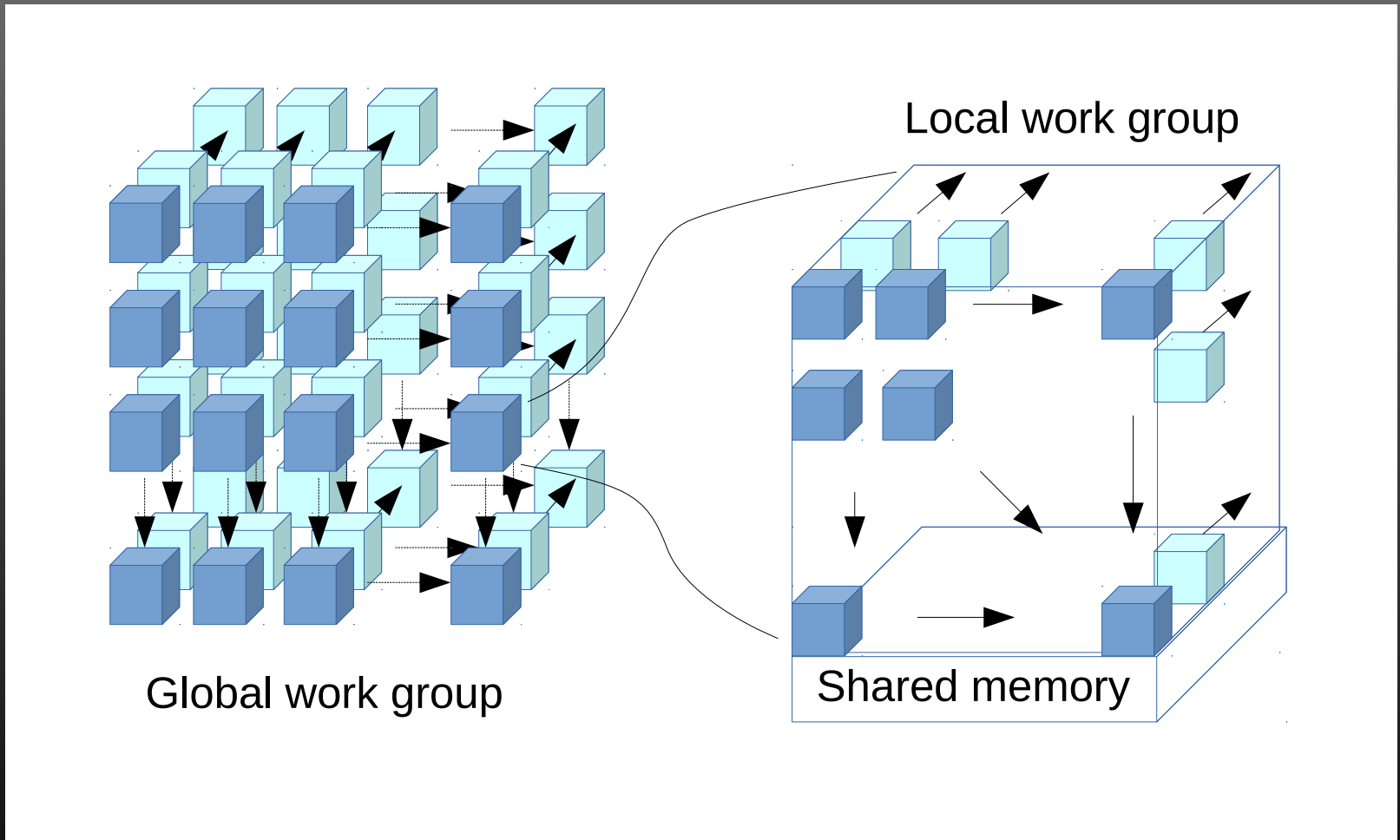
```
...
if (...) {
  ...
  a=a*c; ←
  ...
}
...
```

```
...
if (...) {
  ...
  a=a*c; ⊗
  ...
}
...
```

```
...
if (...) {
  ...
  a=a*c; ←
  ...
}
...
```

```
...
if (...) {
  ...
  a=a*c; ←
  ...
}
...
```

Compute shader : structure



Compute shader : limites

- GL_MAX_COMPUTE_WORK_GROUP_COUNT
  (min value : 65535x65535x65535)
- GL_MAX_COMPUTE_WORK_GROUP_SIZE
  (min value : 1024x1024x64)
- GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS
  (min value : 1024)

Compute shader : un exemple de limites

- GL_MAX_COMPUTE_WORK_GROUP_COUNT
  (min value : 2147483647x65535x65535)
- GL_MAX_COMPUTE_WORK_GROUP_SIZE
  (min value : 1536x1024x64)
- GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS
  (min value : 1536)

Compute shader : Création

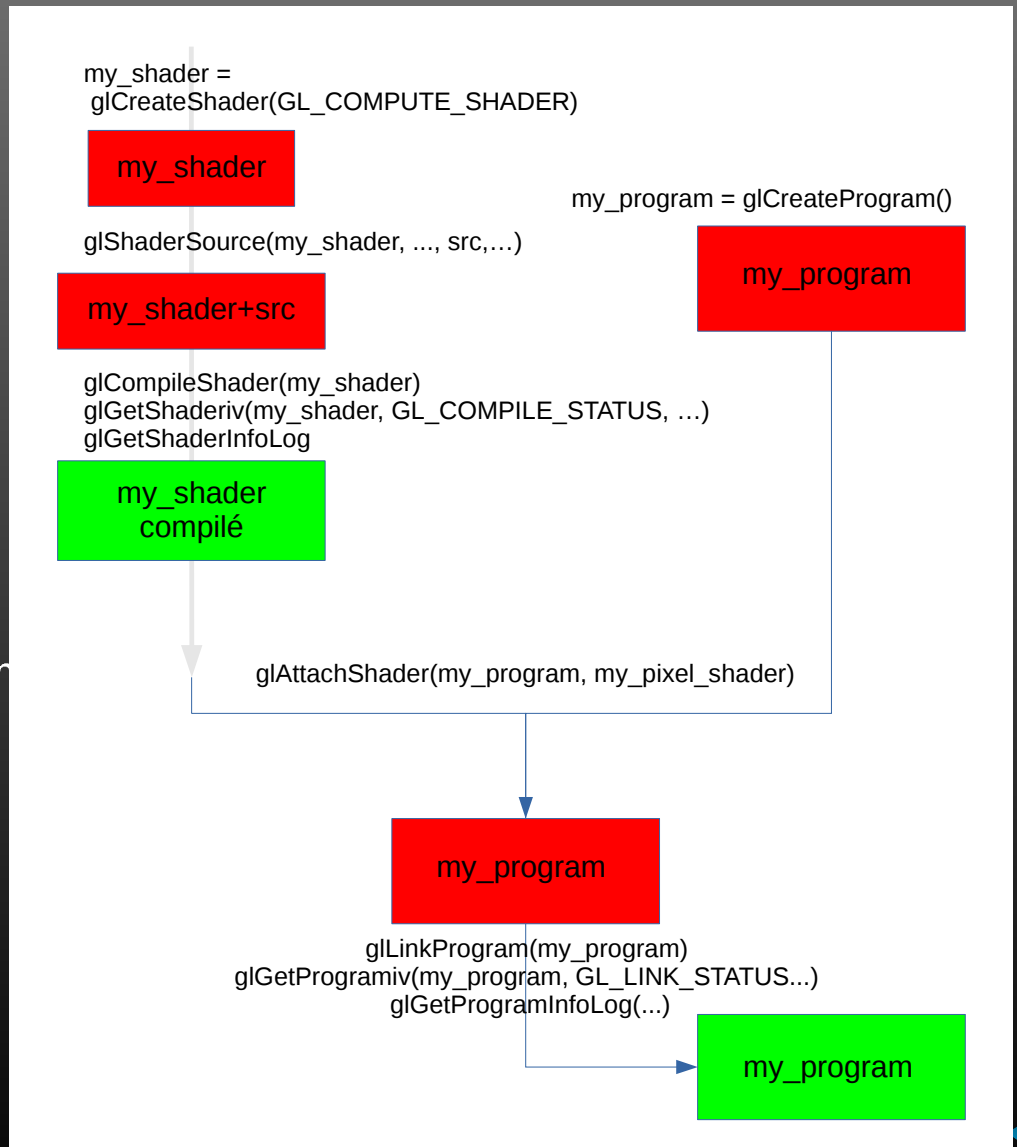
- Langage : GLSL
- Exécution : sur la carte vidéo

Il faut donc compiler le shader avant de pouvoir l'executer :

- Création : glCreateShader()
- Chargement des sources et compilation : glShaderSource()+glCompileSh

Puis l'encapsuler dans un programme :

- Création : glCreateProgram()
- Rattachement des shaders et édition de liens : glAttachShader()+glLinkProgram()



Compute shader : Éxecution

- glUseProgram()
- glDispatchCompute()

Compute shader : Éxecution

Comme une instance d'un *fragment shader* peut savoir de quel *fragment* il s'agit, chaque instance peut savoir qui elle est :

- en absolu : gl_GlobalInvocationID
- localement : gl_LocalInvocationID
- ...

Mais elle ne sait pas qui a déjà été exécuté et qui ne l'est pas...

Compute shader : Retour sur un moteur de particules

Une application possible : un moteur de particules

- Le compute shader met à jour les particules

Avantages :

- Les données des particules ne quittent jamais la *VRAM*
- Libère le CPU

Compute shader : Retour sur un moteur de particules

Les données sont dans un buffer, vu d'abord comme un *SSBO*.

```
glGenBuffers(1, &particule_ssbo_id);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, particule_ssbo_id);
glBufferData(GL_SHADER_STORAGE_BUFFER, sizeof(struct particule)*NB_PARTICULES,
...
```

Compute shader : Moteur de particules, un exemple

Les données sont dans un buffer, vu d'abord comme un *SSBO*.

```
#version 450

(...)

layout (local_size_x = 1024) in;

struct Particule {
  vec3 pos;
  float size;
  vec3 pad;
  int color_index;
};

layout(std430, binding = 1) buffer particule_pos_buffer
{
  Particule particules[NB_PARTICLES];
};
```

Compute shader : Moteur de particules, un exemple

Ajout de règles pour initialiser/mettre à jour les particules à chaque itération

```
void init_particule(int particule_index) {
  float r = rand(...);
  float y = rand(...);
  float theta = rand(...);
  float c = rand(...);
  particules[particule_index].color_index = (MAX_COLOR-1-int(c * 50.0));
  particules[particule_index].pos.x = r*0.1*cos(theta * 2.0*M_PI)+(rand(...))*20
  particules[particule_index].pos.y = -y*10.0-10.0;
  particules[particule_index].pos.z = r*0.1*sin(theta * 2.0*M_PI);

  particules[particule_index].size = 0.1;
}

void update_particule(int i) {
  particules[i].color_index=particules[i].color_index-1;
  if (particules[i].color_index<...) {
    init_particule(i);
  } else {
    float x = rand(...);
    float y = rand(...);
    float z = rand(...);

    particules[i].pos.x+=(x-0.5)*1.0;
    particules[i].pos.y+=y;
    particules[i].pos.z+=(z-0.5)*1.0+0.05;
    particules[i].size += 0.05;
  }
}
```

Compute shader : Moteur de particules, un exemple

Chaque instance met à jour une particule :

```
void main() {
  update_particule(int(gl_GlobalInvocationID.x));
}
```

ou un groupe de particules :

```
void main() {
  for(...) {
    update_particule(int(gl_GlobalInvocationID.x*...+...));
  }
}
```

Compute shader : Moteur de particules, un exemple

Mise à jour :

```
glUseProgram(compute_program);

glDispatchCompute(part_nb, 1, 1);

glMemoryBarrier(GL_ALL_BARRIER_BITS);
```

Compute shader : Moteur de particules, un exemple

Rendu : Les données du buffer, sont maintenant vues comme un *VBO*.

```
glBindBuffer(GL_ARRAY_BUFFER, particule_buf);

glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 7*sizeof(GLfloat)+1*sizeof(GL

glVertexAttribIPointer(2, 1, GL_INT, 7*sizeof(GLfloat)+1*sizeof(GLint),
(GLvoid *)(7*sizeof(GLfloat)));

glVertexAttribPointer(3, 1, GL_FLOAT, GL_FALSE, 7*sizeof(GLfloat)+1*sizeof(GL
(GLvoid *)(3*sizeof(GLfloat)));
```

Compute shader : Moteur de particules, un exemple

```
glUseProgram(display_program);

glBindVertexArray(main_vao_id);

glDrawArrays(GL_POINTS, 0, NB_PARTICULES);
```

Compute Shader : Un exemple de calcul

Compute Shader : Un exemple de calcul

Diminuer l'intensité d'une image :

```
#version 430 core

layout (local_size_x = 16, local_size_y = 16) in;

layout (rgba8ui, binding = 0) readonly uniform uimage2D input_image;
layout (rgba8ui, binding = 1) writeonly uniform uimage2D output_image;


void main(void) {
  ivec2 pos = ivec2(gl_GlobalInvocationID.xy);
  uvec4 result = imageLoad(input_image, pos)/2;
  imageStore(output_image, pos, result);
}
```

Compute Shader : Un autre exemple de calcul

Compute Shader : Un autre exemple de calcul

Filtrer une image :

```
#version 430 core

layout (local_size_x = 1024) in;

layout (rgba32f, binding = 0) uniform image2D input_image;
layout (rgba32f, binding = 1) uniform image2D output_image;

void main(void)
{

  ivec2 pos = ivec2(gl_GlobalInvocationID.xy);
  vec4 result = (imageLoad(input_image, pos) +
  imageLoad(input_image, min(pos.x + 1, 1023))+
  imageLoad(input_image, max(pos.x - 1, 0)))/3.0;

  imageStore(output_image, pos.yx, result);
}
```

Compute Shader : Un autre exemple de calcul

```
glDispatchCompute(1, 1024, 1);
(...)
//  glMemoryBarrier(GL_SHADER_IMAGE_ACCESS_BARRIER_BIT);
(...)
glDispatchCompute(1, 1024, 1);
```

Compute Shader : Un autre exemple de calcul

Pour bien comprendre :

```
layout (local_size_x = 1024) in;
```

On a un *local work group* qui traite les lignes (ici en constante de 1024 pixels). Donc 1024 instances par *local work group*

```
glDispatchCompute(1, 1024, 1);
```

On a un *global work group* de 1024 *local work group* (verticalement) et qui délègue le travail par ligne.

Compute Shader : Un autre exemple de calcul

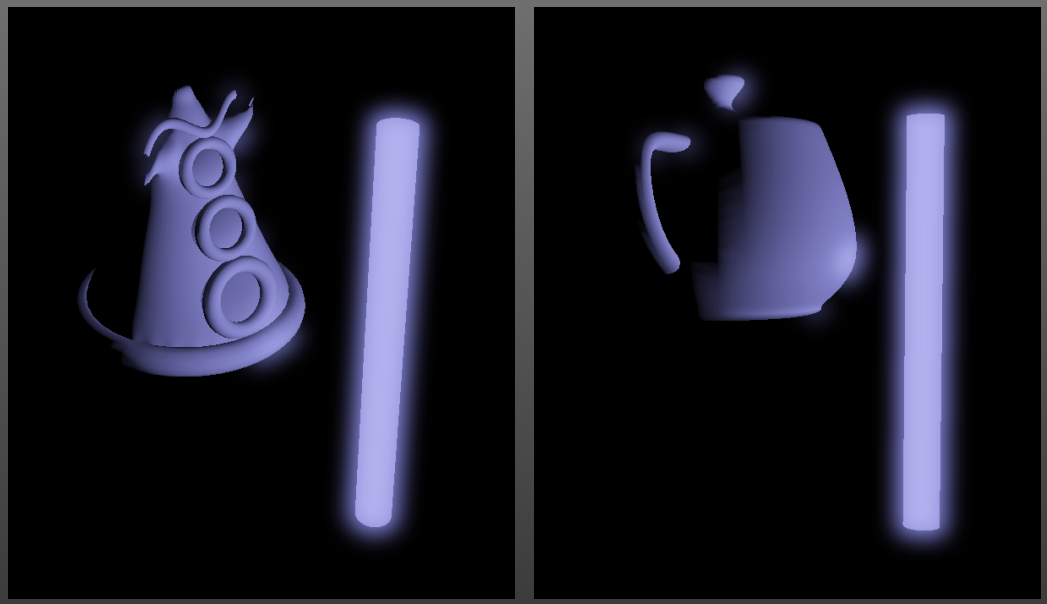
Filtrer une image un peu plus vite :

```
#version 430 core

layout (local_size_x = 1024) in;

layout (rgba32f, binding = 0) uniform image2D input_image;
layout (rgba32f, binding = 1) uniform image2D output_image;

shared vec4 scanline[1024];

void main(void)
{

  ivec2 pos = ivec2(gl_GlobalInvocationID.xy);
  scanline[pos.x] = imageLoad(input_image, pos);
  barrier();


  vec4 result = (scanline[pos] + scanline[min(pos.x + 1, 1023)] + scanline[r

  imageStore(output_image, pos.yx, result);
}
```

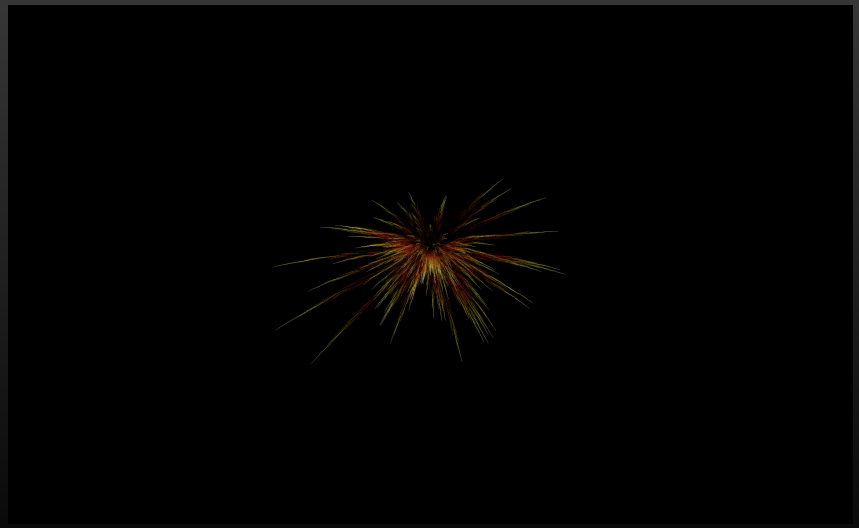
Compute Shader : Un autre exemple de calcul

```
glDispatchCompute(1, 1024, 1);
(...)
//  glMemoryBarrier(GL_SHADER_IMAGE_ACCESS_BARRIER_BIT);
(...)
glDispatchCompute(1, 1024, 1);
```

Compute Shader : Conclusion



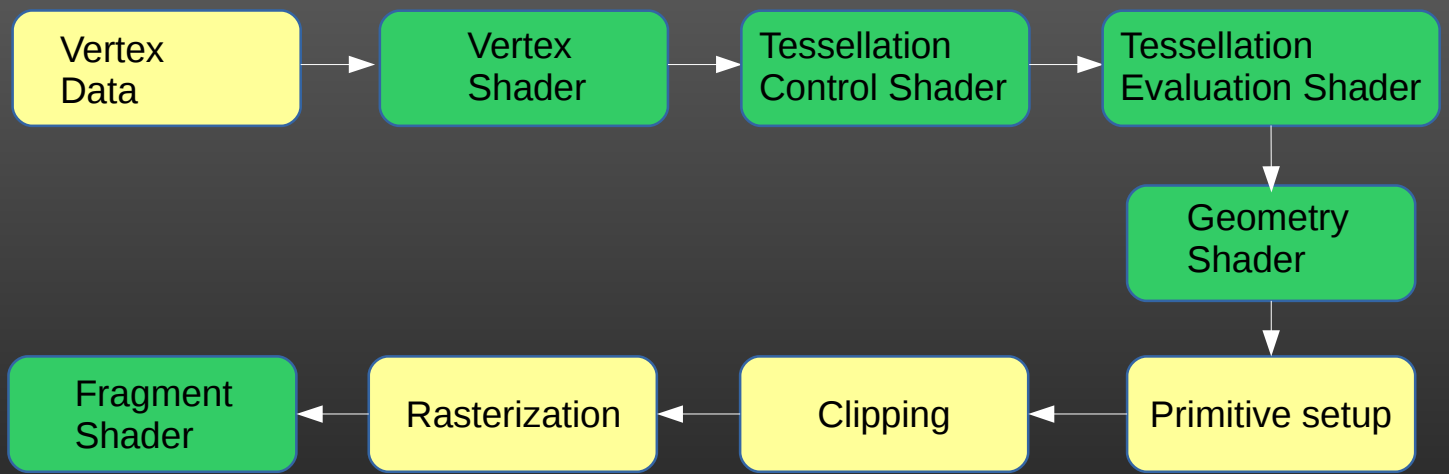
blooming effect



Particules

Geometry Shader

Geometry Shader



Geometry Shader : input

L'application envoie des primitives :

- *Points, lines, line strip, loop line, triangles, triangle strip, triangle fan*

Mais aussi :

- *Lines with adjacency, line strip with adjacency, triangles with adjacency, triangle strip with adjacency*

Le *geometry shader* reçoit :

- *Point, Line, Triangle*
- *Line with adjacency, Triangle with adjacency*

Geometry Shader : output

Le *geometry shader* renvoie :

- *Points, LineStrips, TriangleStrips*

Important :

- Il n'y a pas de lien entre le type des éléments reçus et le type des éléments émis
- Le *Geometry Shader* peut renvoyer 0, 1 ou plusieurs éléments

Le *geometry shader* peut donc :

- Ajouter des primitives sur le pipeline (ou intercepter)
- Changer le type de primitives sur le pipeline

Geometry Shader : données

Geometry Shader : données

- Déclaration des primitives (entrées/sorties) :

```
layout( triangles ) in;
layout( line_strip, max_vertices=10 ) out;
```

Geometry Shader : données

- Déclaration des primitives (entrées/sorties) :

```
layout( triangles ) in;
layout( line_strip, max_vertices=10 ) out;
```

- Compléments :

```
in vec3 gs_normal[3];

out vec3 color;
```

Geometry Shader : données

- Déclaration des primitives (entrées/sorties) :

```
layout( triangles ) in;
layout( line_strip, max_vertices=10 ) out;
```

- Compléments :

```
in vec3 gs_normal[3];

out vec3 color;
```

- Accès

```
gl_in[0].gl_Position
gs_normal[0]
gl_in[1].gl_Position
gs_normal[1]
gl_in[2].gl_Position
gs_normal[3]
```

Geometry Shader : données

- Déclaration des primitives (entrées/sorties) :

```
layout( triangles ) in;
layout( line_strip, max_vertices=10 ) out;
```

- Compléments :

```
in vec3 gs_normal[3];

out vec3 color;
```

- Accès

```
gl_in[0].gl_Position
gs_normal[0]
gl_in[1].gl_Position
gs_normal[1]
gl_in[2].gl_Position
gs_normal[3]
```

- *built in input variables* :

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

Geometry Shader : données

- Autres déclarations

```
uniform mat4 model_view_matrix;
uniform mat4 projection_matrix;
```

Geometry Shader : code

- Déclarations

```
...
layout( line_strip, max_vertices=10 ) out;
...


...
out vec3 color;
...
```

- Exécution

```
color = ...;
gl_Position = ...;
EmitVertex();

color = ...;
gl_Position = ...;
EmitVertex();

EndPrimitive(); //optionnel en fin de shader
```

Geometry Shader : Sorties

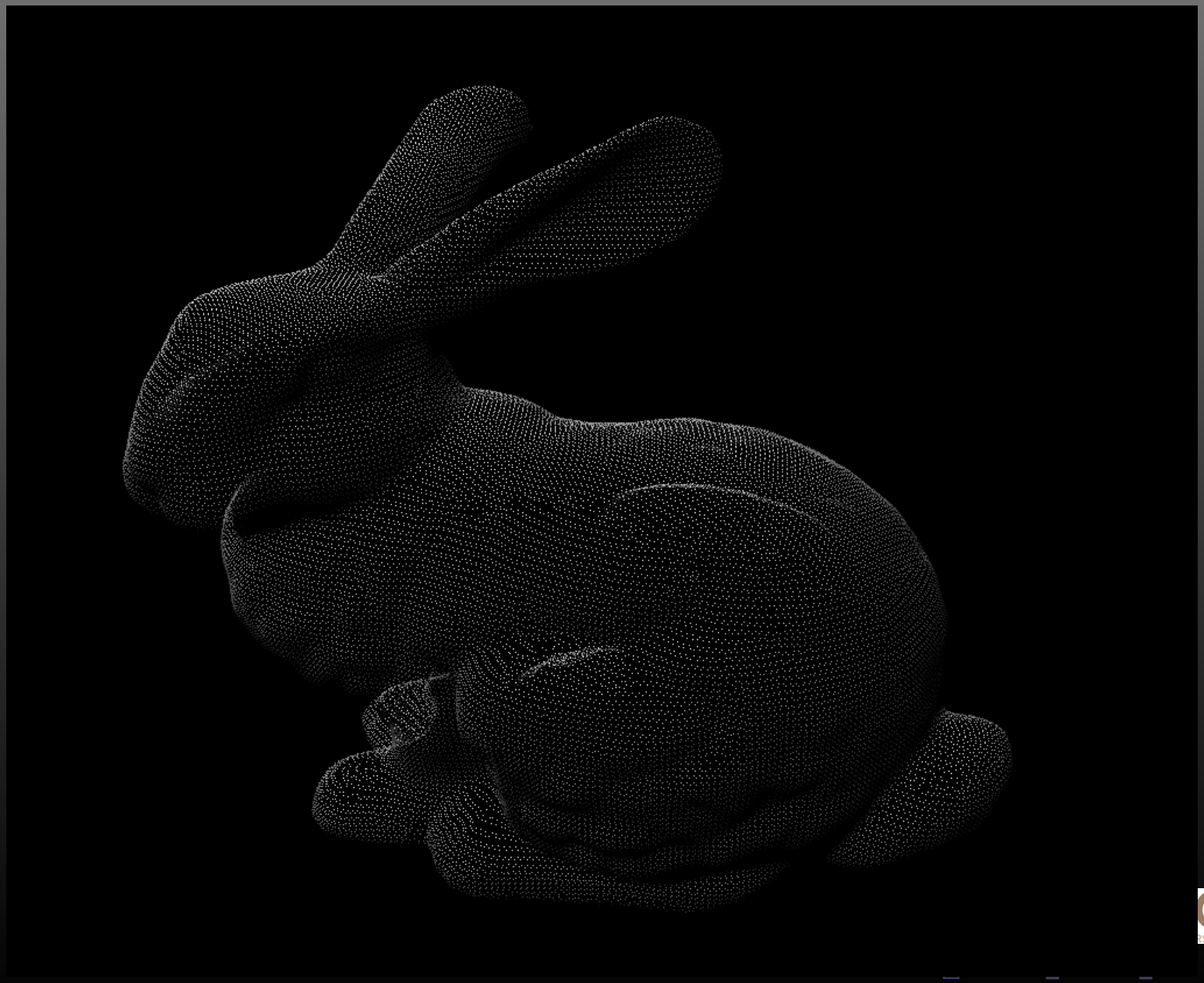
- Sorties

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

Geometry Shader : un exemple de conversion en points

```
#version 450

layout( triangles )  in;
layout( points, max_vertices=1 ) out;

in vec3 gs_normal[3];
out vec3 color;

uniform mat4 model_view_matrix;
uniform mat4 projection_matrix;
uniform vec3 object_color;
uniform vec3 light_position;

void main {
  vec4 origin = (gl_in[0].gl_Position + gl_in[1].gl_Position + gl_in[2].gl_
  vec3 normal_inter = (gs_normal[0] + gs_normal[1] + gs_normal[2]) / 3.0;

  vec3 light_dir = normalize(light_position-origin.xyz);

  color = object_color*dot(light_dir, normal_inter);
  gl_Position = projection_matrix * model_view_matrix * origin;
  EmitVertex();

  EndPrimitive();
}
```

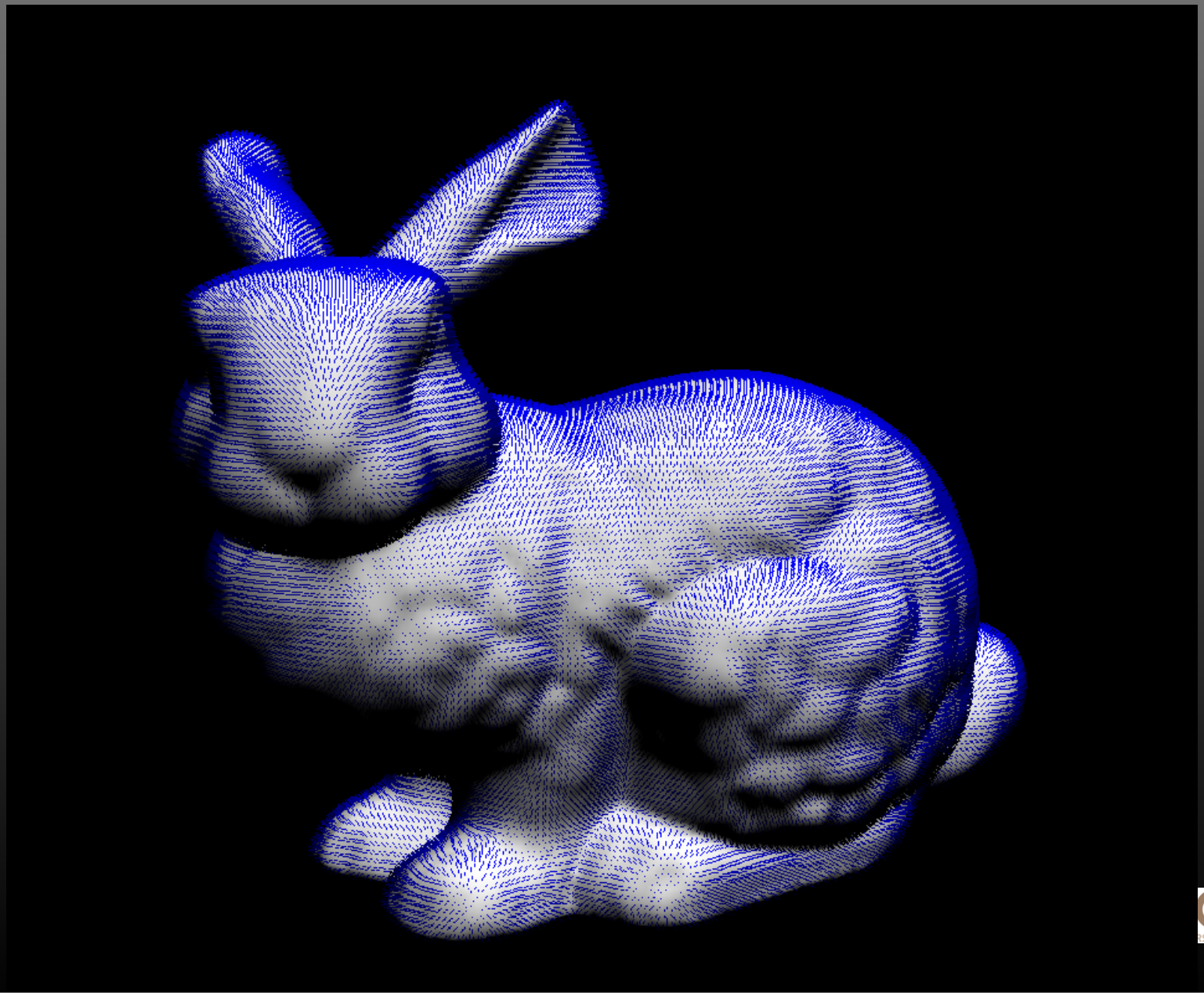
Geometry Shader : un exemple de conversion en points



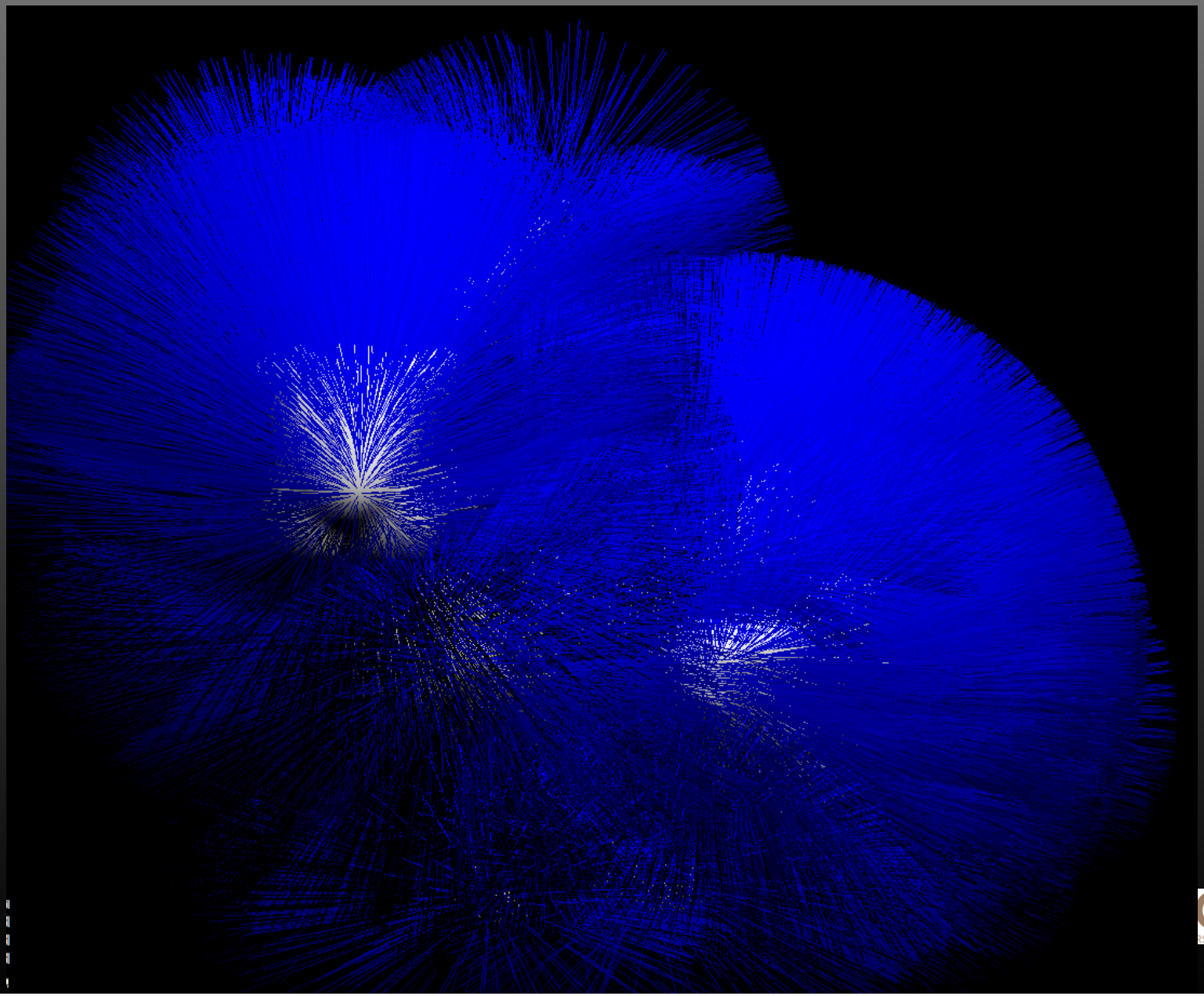
Geometry Shader : un exemple de visualisation des normales

```
#version 450

layout( triangles ) in;
layout( line_strip, max_vertices=2 ) out;

in vec3 gs_normal[3];

uniform mat4 model_view_matrix;
uniform mat4 projection_matrix;
uniform vec3 object_color;
uniform vec3 light_position;
uniform float normal_length;

out vec3 color;


void main() {
  vec4 origin = (gl_in[0].gl_Position + gl_in[1].gl_Position + gl_in[2].gl_Posi
  vec3 normal_inter = (gs_normal[0] + gs_normal[1] + gs_normal[2]) / 3.0;

  vec3 light_dir = normalize(light_position-origin.xyz);
  color = object_color*clamp(dot(light_dir, normal_inter), 0.0, 1.0);

  gl_Position = projection_matrix * model_view_matrix * (origin);
  EmitVertex();
  gl_Position = projection_matrix * model_view_matrix * (origin+normal_length*ve
  EmitVertex();

  EndPrimitive();
}
```

Geometry Shader : un exemple de visualisation des normales



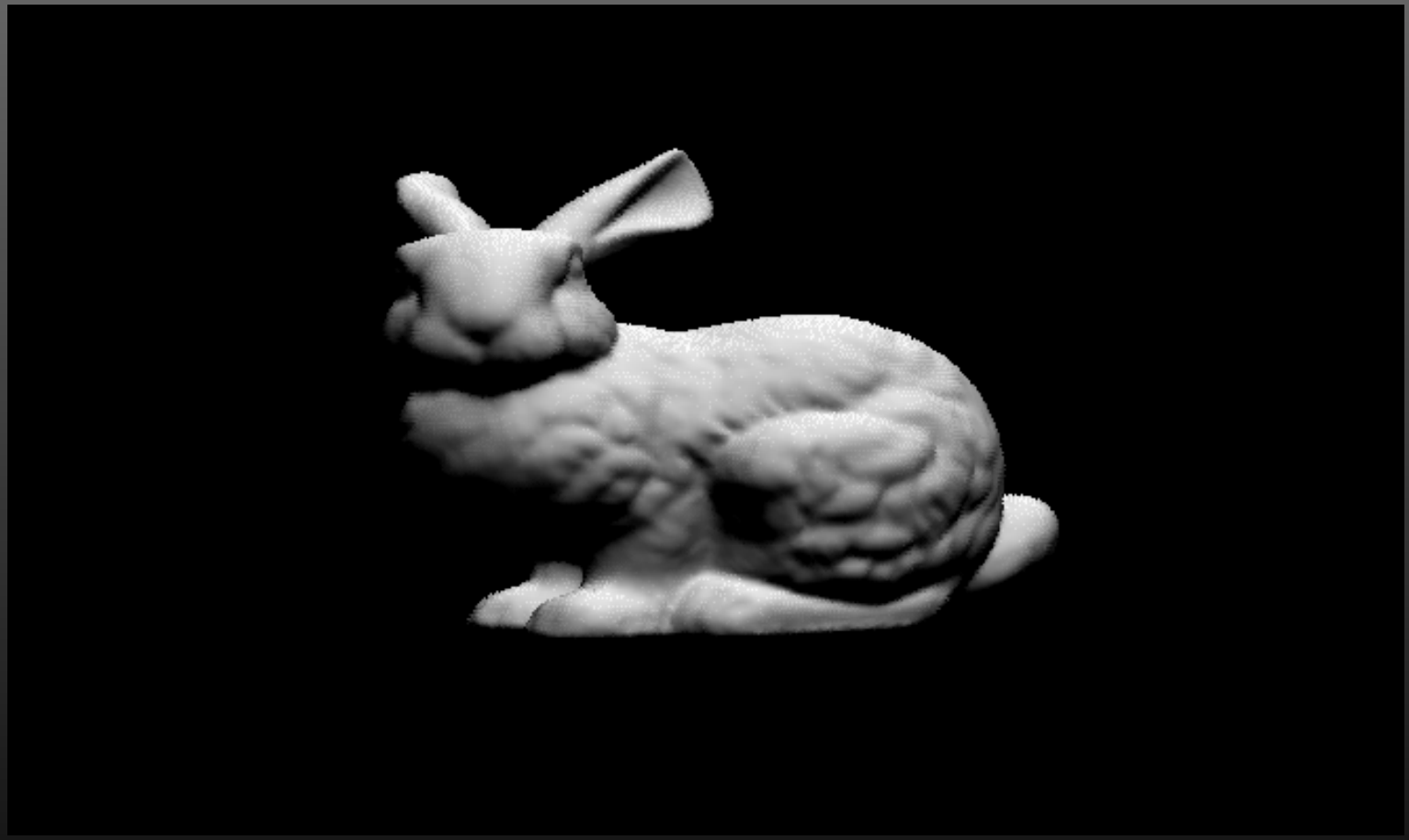
Geometry Shader : un lapin porc-épic



Geometry Shader : un lapin angora

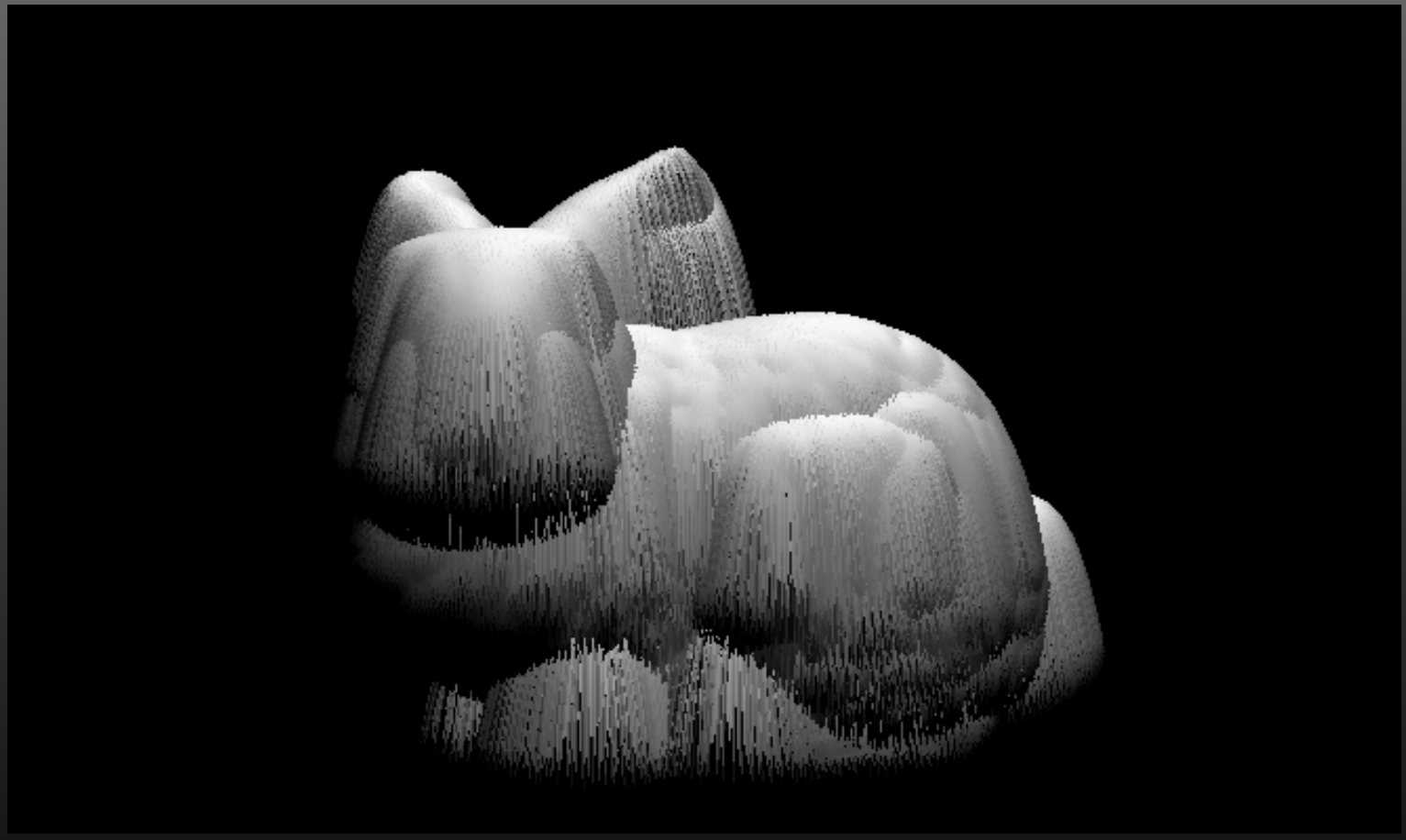
```
layout( line_strip, max_vertices=10 ) out;

(...)

for(float t = 0 ; t < fur_length ; t += step) {

  vec3 p1 = normal * t + origin.xyz + vec3(0.0, -1.0/2.0*t*t*9.81*9.8, 0.0);
  vec3 p2 = normal * t+step + origin.xyz + vec3(0.0, -1.0/2.0*(t+step)*(t+st

  color = ...

  gl_Position = projection_matrix * model_view_matrix * vec4(p1,1.0);
  EmitVertex();
}

(...)
```

Geometry Shader : un lapin angora



Geometry Shader : un lapin angora

Geometry Shader : un lapin angora



Geometry Shader : Quand passer dans le repère normalisé ?

```
uniform mat4 model_view_matrix;
uniform mat4 projection_matrix;
(...)
gl_position = projection_matrix * model_view_matrix * p;
```

Si c'est fait dans :

- le *Vertex Shader*, c'est fait moins souvent si on raffine le modèle dans le *Geometry Shader* mais il faut s'assurer que les calculs restent vrai dans le ce repère.
- le *Geometry Shader*, c'est fait plus souvent si on raffine le modèle mais on peut travailler dans le repère monde.

Geometry Shader : Un input un peu particulier

Il est possible de fournir des informations supplémentaires dans les *VBOs* à savoir les formes adjacentes

- Lines with Adjacency (GL_LINES_ADJACENCY)
- Line strip with adjacency (GL_LINE_STRIP_ADJACENCY)
- Triangles with adjacency (GL_TRIANGLES_ADJACENCY)
- Triangle strip with adjacency (GL_TRIANGLE_STRIP_ADJACENCY)

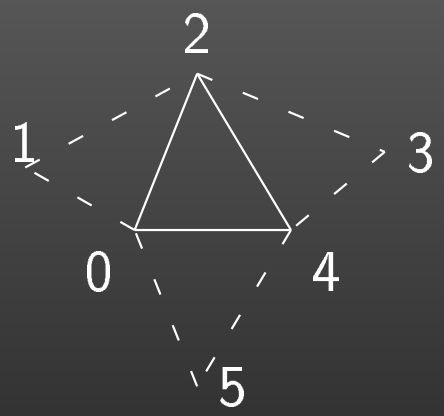
Geometry Shader : Un input un peu particulier

- Lines with Adjacency (GL_LINES_ADJACENCY)

- Line strip with adjacency (GL_LINE_STRIP_ADJACENCY)

Geometry Shader : Un input un peu particulier

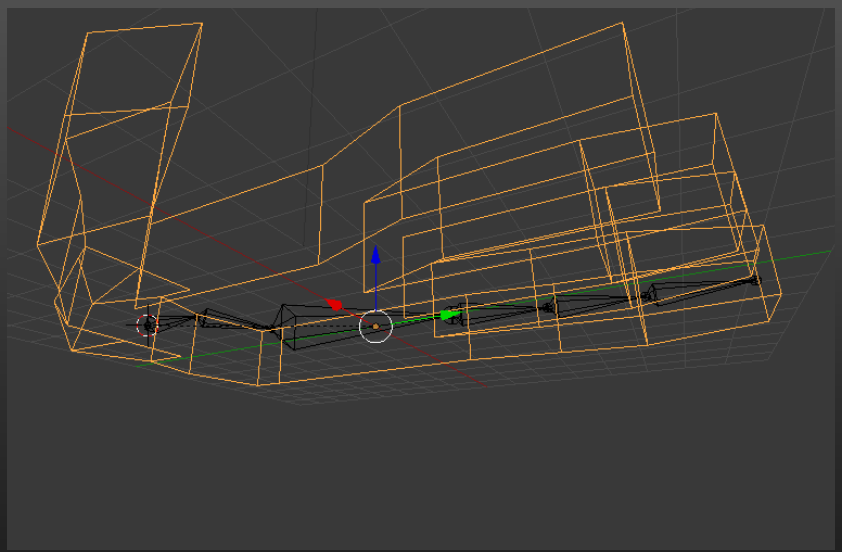
- Triangles with adjacency (GL_TRIANGLES_ADJACENCY)

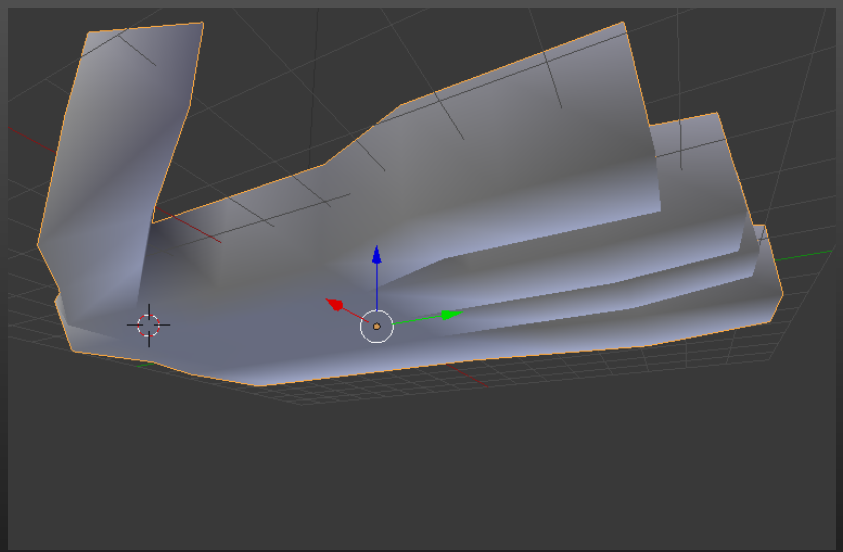


- Triangle strip with adjacency (GL_TRIANGLE_STRIP_ADJACENCY)

Geometry Shader : Un input un peu particulier

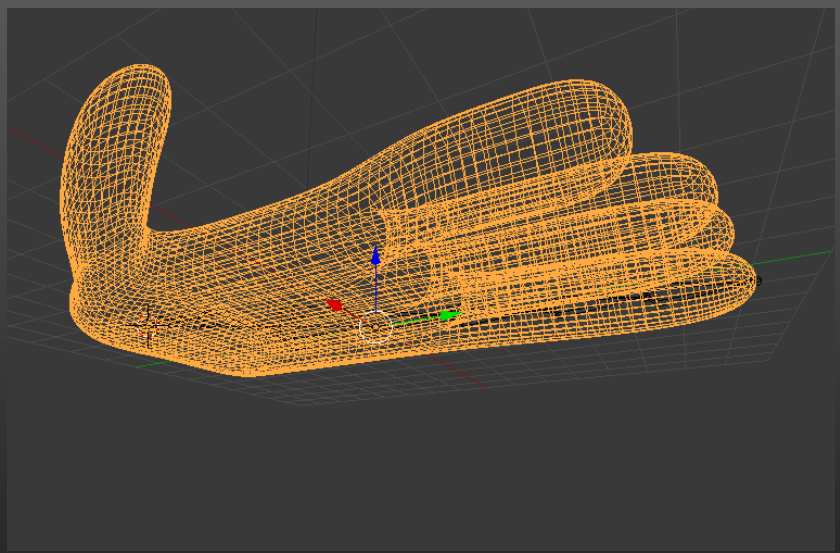
Permet par exemple le lissage : On peut avoir un modèle très simplifié (*110 vertices*)

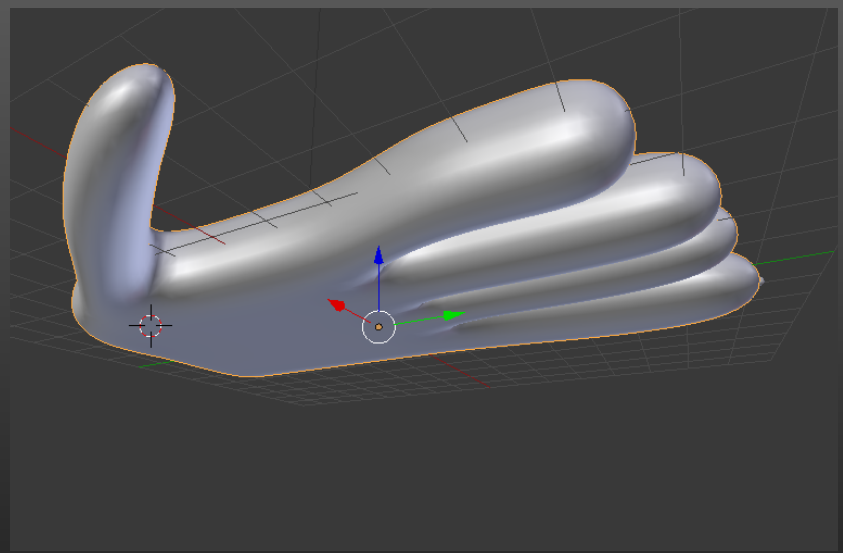




Geometry Shader : Un input un peu particulier

Permet par exemple le lissage : Que l'on peut raffiner en temps réel (6 914 *vertices*).

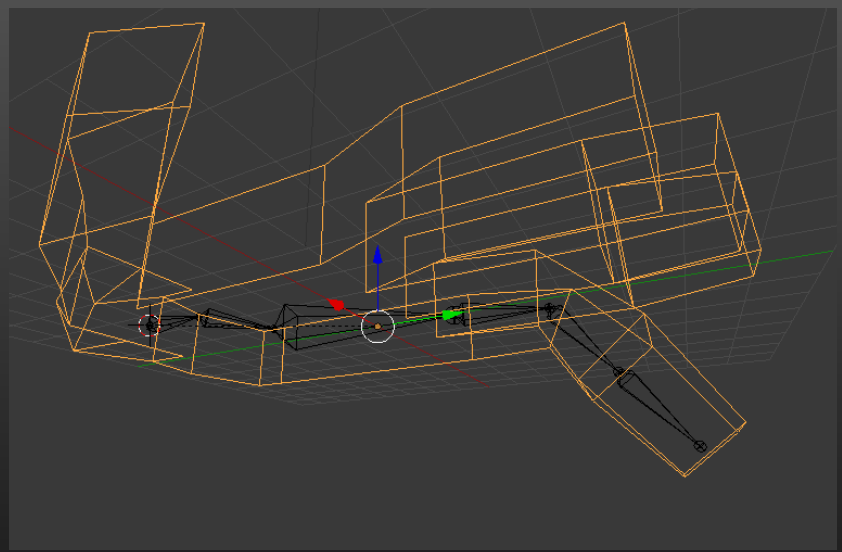


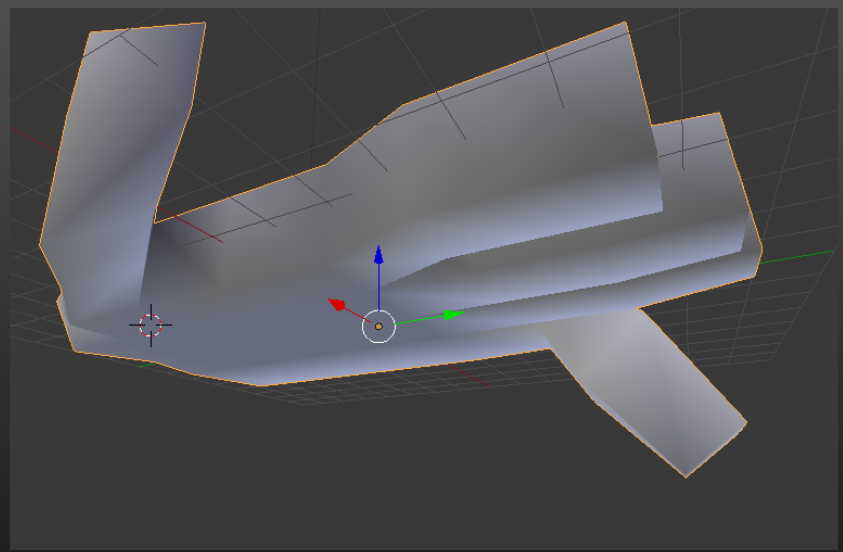


On a souvent besoin de connaître le voisinage pour pouvoir lisser correctement. (Ce lissage peut être dépendant de la distance à l'observateur).

Geometry Shader : Un input un peu particulier

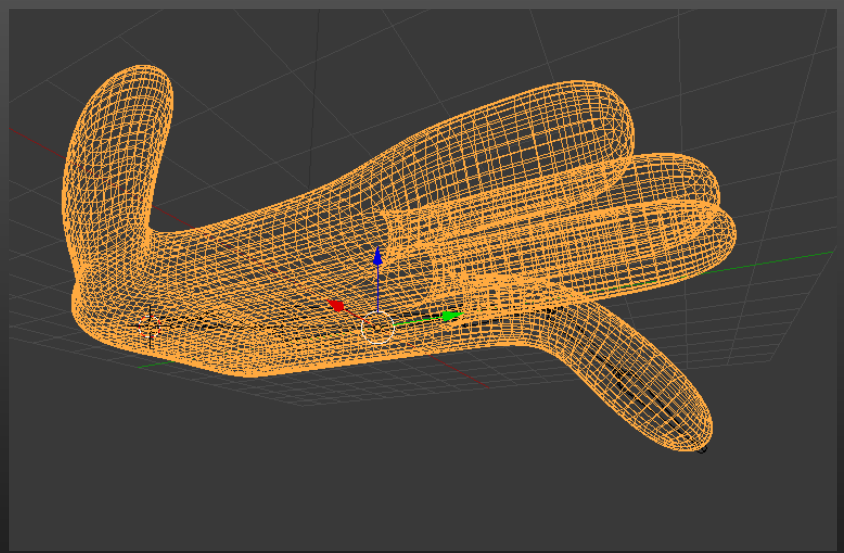
Permet par exemple le lissage : Pour animer le modèle, on peut animer le modèle simplifié (que quelques sommets).

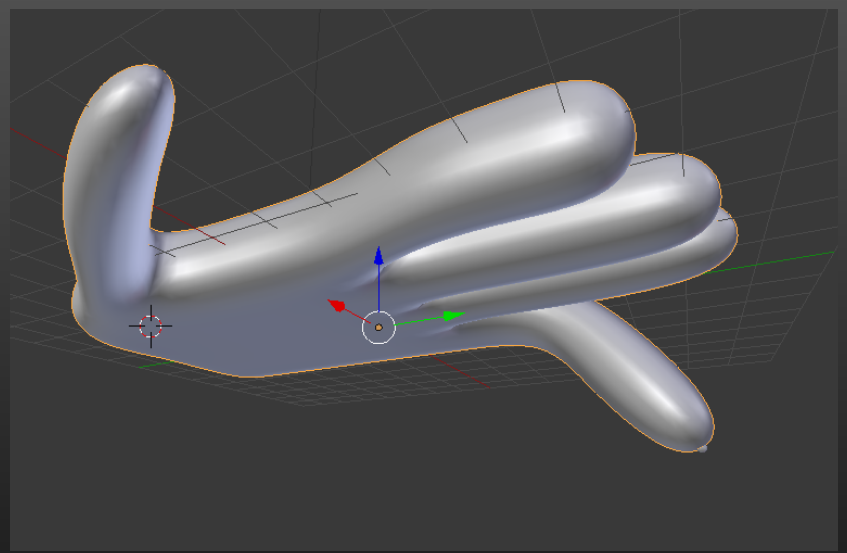




Geometry Shader : Un input un peu particulier

Permet par exemple le lissage : Et on obtient une animation du modèle complet.





Geometry Shader : Un input un peu particulier

- Certains lissages préservent le type de primitive, d'autres ont besoin de primitives de différente sortes (*quad*, triangle...), toutefois le *Geometry Shader* ne peut sortir qu'un type à la fois, on découpera donc les primitives pour n'avoir qu'un seul type.
- Ce lissage peut être fait en temps réel dans le *Geometry Shader* un utilisant les primitives adjacentes.

Geometry Shader : conclusion

Pas obligatoire mais permet de gagner du temps :

- Permet d'enrichir le model (lissage...)
- Permet de changer les primitives (Bezier Patch...)
- Permet l'ajout d'effets (herbe, fourrure...)
- Permet l'ajout d'autres effets (explosions...)
- ...

Note : il est possible de boucler sur le *geometry shader*.

Tessellation Shaders

Tessellation Shaders

- L'objectif des *tessellation shaders* et de subdiviser les primitives directement sur le pipeline graphique.

Tessellation Shaders

- L'objectif des *tessellation shaders* et de subdiviser les primitives directement sur le pipeline graphique.
- Les *tessellation shaders* ont plein de cas d'applications :
  - subdivisions adaptatives/*mesh refinement*
  - geometric compression
  - application des *displacement maps*
  - ...

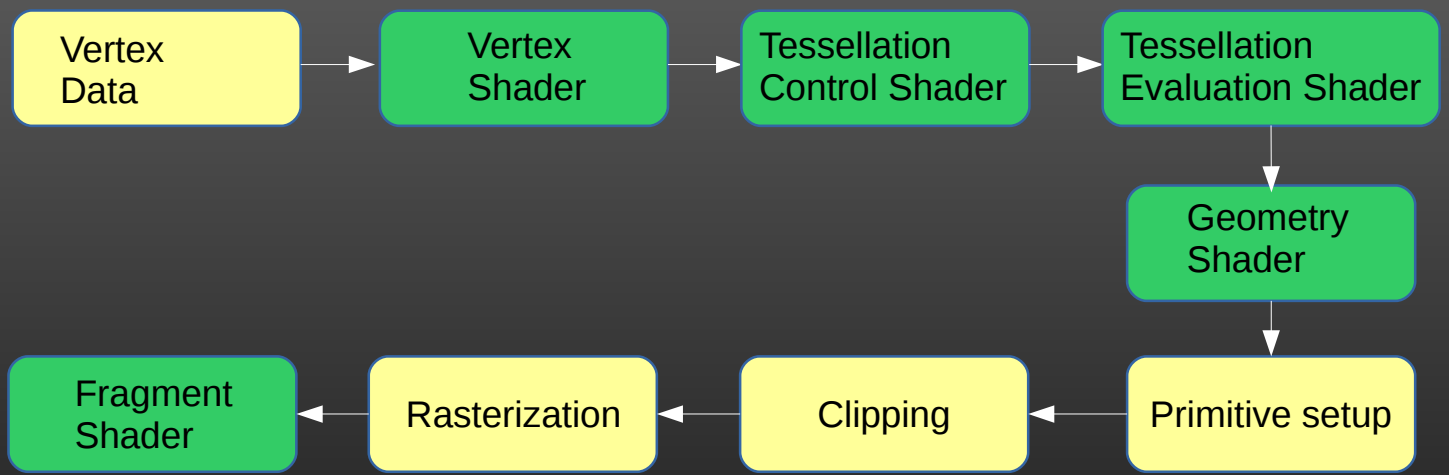
Tessellation Shaders

- L'objectif des *tessellation shaders* et de subdiviser les primitives directement sur le pipeline graphique.
- Les *tessellation shaders* ont plein de cas d'applications :
  - subdivisions adaptatives/*mesh refinement*
  - geometric compression
  - application des *displacement maps*
  - ...
- Introduits à partir d'OpenGL 4.1

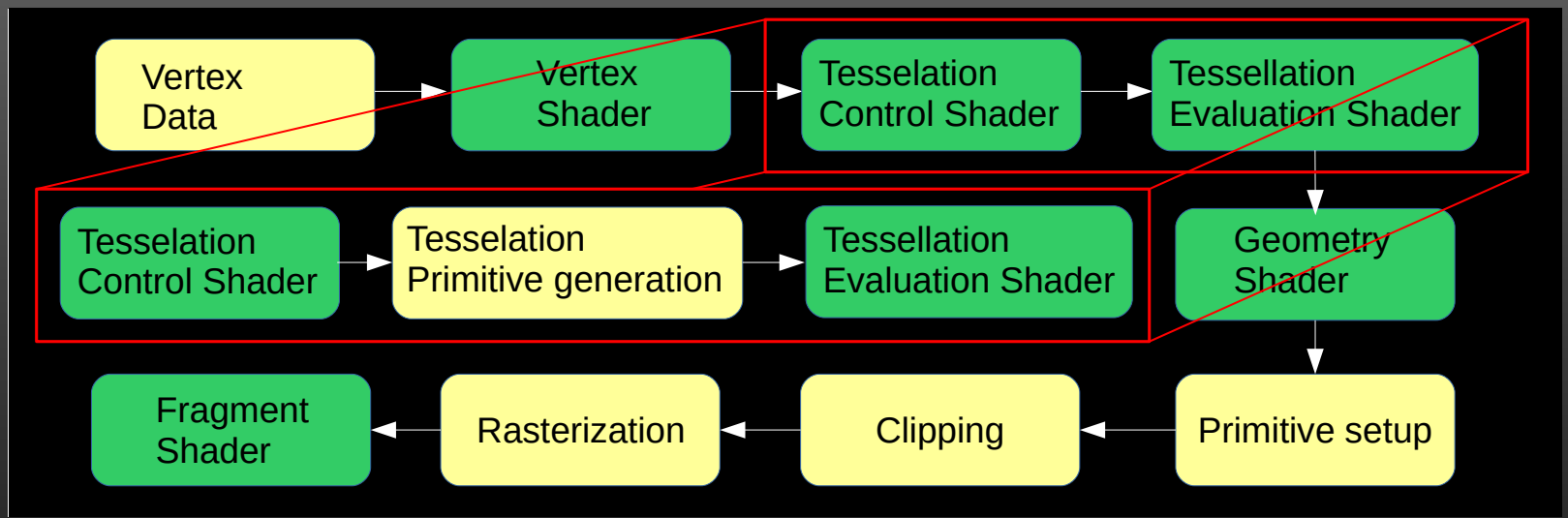
Tessellation Shaders

- L'objectif des *tessellation shaders* et de subdiviser les primitives directement sur le pipeline graphique.
- Les *tessellation shaders* ont plein de cas d'applications :
  - subdivisions adaptatives/*mesh refinement*
  - geometric compression
  - application des *displacement maps*
  - ...
- Introduits à partir d'OpenGL 4.1
- Peuvent subdiviser des lignes (`isolines`), des triangles (`triangles`) ou des quadrilatères (`quads`).

Tessellation Shaders



Tessellation Shaders



Tessellation Shaders

- *Tessellation control* : Définie le niveau de *tessellation* et prépare le *patch*
- *Tessellation primitive generation* : Effectue le découpage
- *Tessellation evaluation* : Convertit les sommets abstraits issus du *Tessellation primitive generation* en sommets réels

Tessellation Shaders

- *Tessellation control* : Programmable
- *Tessellation primitive generation* : Fixe
- *Tessellation evaluation* : Programmable

Tessellation Shaders

- *Tessellation control* : Exécuté une fois par sommet (généré) dans le *patch*
- *Tessellation evaluation* : Exécuté une fois par sommet issu de la *tessellation*

Tessellation Shaders : *Inputs*

- Emission d'un *patch* sur le pipeline :
  - Le *draw* est invoqué avec la primitive : `GL_PATCHES`
  - On peut contrôler la taille du *patch* émis dans le pipelie avec `glPatchParameteri(GL_PATCH_VERTICES, GLint value);`
- La taille maximum d'un *patch* émis sur le pipeline est d'au moins 32 éléments (c.f. `GL_MAX_PATCH_VERTICES`) (mais peut être plus grande suivant la carte vidéo).
- La taille du patch émis sur le pipeline peut être différente du patch qui va effectivement être subdivisé

Tessellation Shaders : *Tessellation control*

Tessellation Shaders : *Tessellation control*

- Objectifs : générer le patch qui va être subdivisé et contrôler le niveau de subdivisions.

Tessellation Shaders : *Tessellation control*

Entrées :

- *built in input variables* :

```
in gl_PerVertex
{
  vec4 gl_Position;
  float gl_PointSize;
  float gl_ClipDistance[];
} gl_in[];
in int gl_PatchVerticesIn;
in int gl_PrimitiveID;
in int gl_InvocationID;
```

Il peut accéder à l'ensemble des sommets du *patch* émis sur le pipeline. Il peut repérer un sommet vis à vis des autres instances à l'aide de `gl_InvocationID` :
`(gl_in[gl_InvocationID].gl_Position;)`

Tessellation Shaders : *Tessellation control*

En sortie : 1/ Il doit contrôler le niveau de subdivisions.

- `gl_TessLevelInner[]` : niveau de subdivision de l'intérieur du *patch*
- `gl_TessLevelOuter[]` : niveau de subdivision du contour du *patch*

Attention à la continuité entre les *patches* !

Tessellation Shaders : *Tessellation control*

En sortie : 2/ Génération du *patch* à subdiviser :

- Il peut accéder à l'ensemble des sommets du *patch* émis sur le pipeline (`gl_in[gl_InvocationID].gl_Position;`)
- Il doit déclarer la taille du *patch* à subdiviser : `layout (vertices=4) out;`
- Si besoin, il peut écrire dans

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

Tessellation Shaders : *Tessellation control*

Cette étape est optionnelle.

Tessellation Shaders : *Tessellation primitive generation*

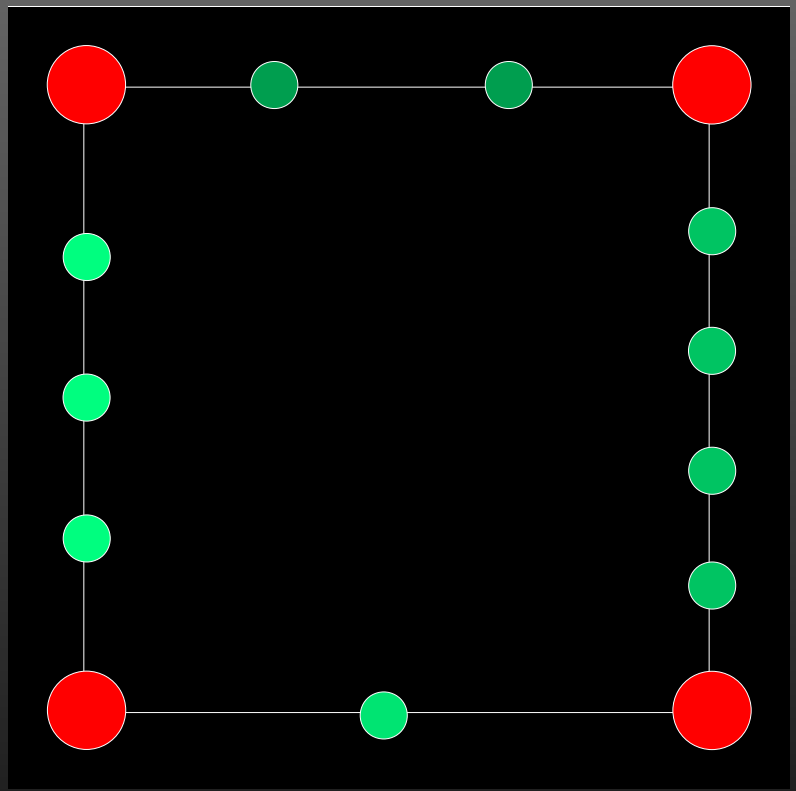
Tessellation Shaders : *Tessellation primitive generation*

- Objectif : générer une subdivision d'un *patch abstrait*

Tessellation Shaders : *Tessellation primitive generation*

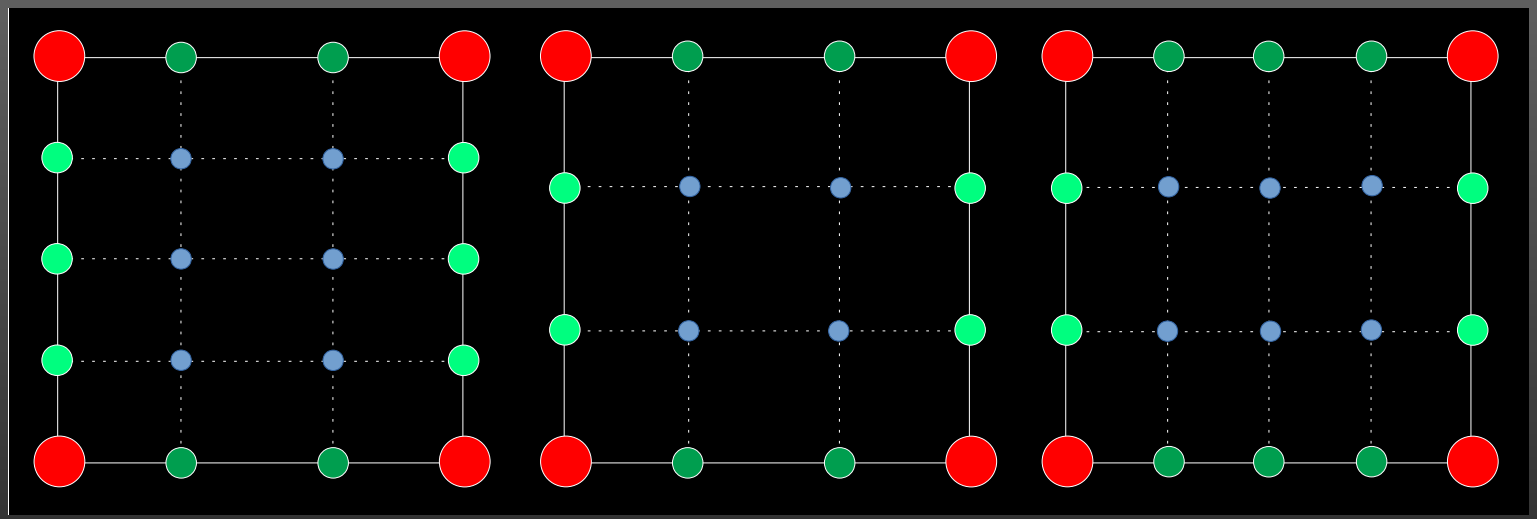
- En fonction de `gl_TessLevelInner[]`, `gl_TessLevelOuter[]` et du choix du type de *patch* (`isoline`, `quad` ou `triangle`), réalise la *tessellation*.
- Travaille dans un repère local au *patch* (travail sur un *patch abstrait*)

Tessellation Shaders : *Tessellation primitive generation*



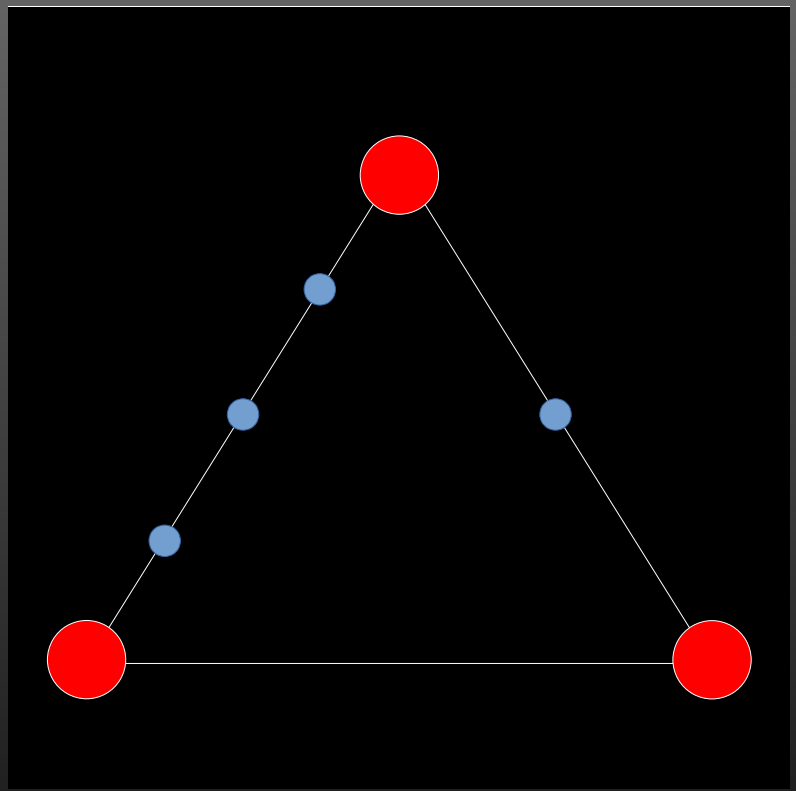
```
gl_TessLevelOuter[] = {4, 2, 5, 3}
```

Tessellation Shaders : *Tessellation primitive generation*



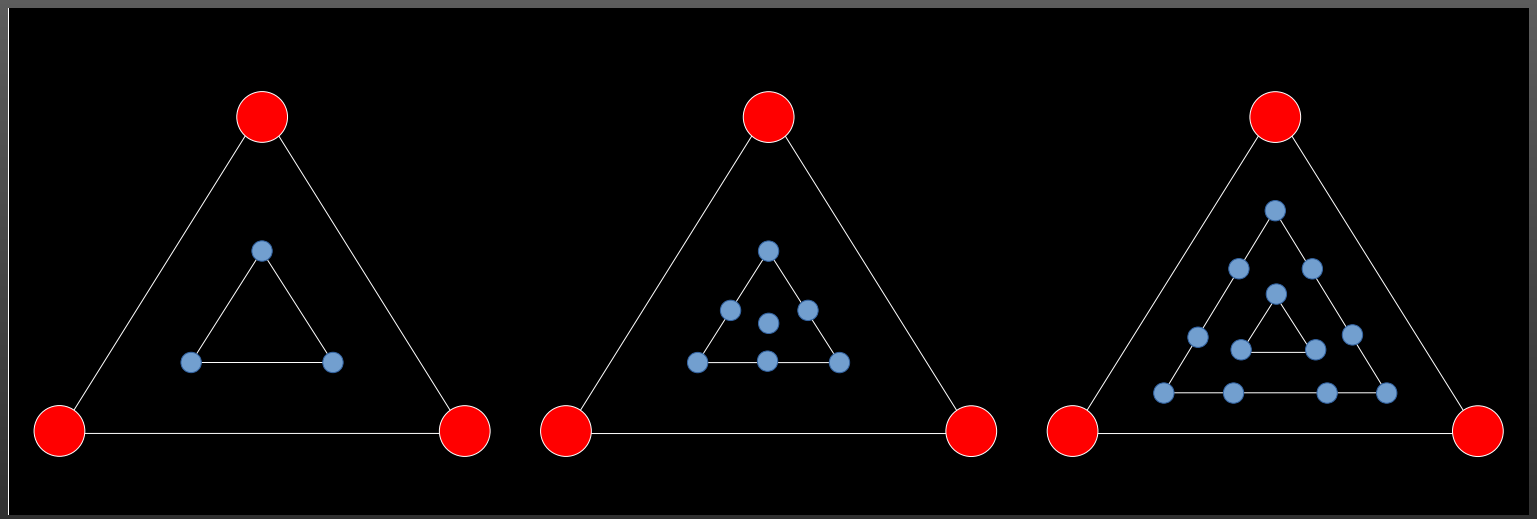
```
gl_TessLevelInner[] = {3, 4},
gl_TessLevelInner[] = {3, 3},
gl_TessLevelInner[] = {4, 3}
```

Tessellation Shaders : *Tessellation primitive generation*



```
gl_TessLevelOuter[] = {4, 1, 2}
```

Tessellation Shaders : *Tessellation primitive generation*



`gl_TessLevelInner[] = {3}`, `gl_TessLevelInner[] = {4}`,
`gl_TessLevelInner[] = {5}`

Tessellation Shaders : *Tessellation primitive generation*

Les coordonnées des sommets sont dans un repère local aux *patches*.

- Pour le triangle : ce sont les coordonnées barycentriques
- Pour le quadrilatère : il y a un axe en x et un axe y

Tessellation Shaders : *Tessellation primitive generation*

- Une fois les sommets définis, le *patch* est *triangulé* mais toujours dans le repère local.
- L'implémentation n'est pas imposées mais la spécification assure des garanties sur le résultat.

Tessellation Shaders : *Tessellation evaluation shader*

Il faut maintenant déduire la forme finale en passant du repère local au repère caméra et faire la projection. C'est le rôle de ce dernier *shader*.

- Il a d'un coté le *patch* de l'autre un *patch abtrait* qui indique comment découper le *patch*.
- invoqué pour chaque sommet, ce *shader* peut accéder au *patch* complet
- le sommet (du *patch abtrait*) qu'il doit traiter est passé dans `gl_TessCoord` (dans le repère local)

Si c'est un `quad`, c'est dans le repère du quadrilatère, si c'est un `triangle`, c'est les coordonnées barycentriques dans le triangle.

Tessellation Shaders : exemple simple

Etape 1 :

- Le *vertex shader* reçoit les coordonnées du *patch*
- Il les transmet au *tessellation control shader*

```
#version 450

layout(location = 1) in vec3 vPosition;

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

Tessellation Shaders : exemple simple

Etape 2 :

- Le *tessellation control shader* reçoit à son tour les coordonnées du *patch*
- Il définit le niveau de tessellation
- Il transmet les coordonnées du *patch* au suivant

```
#version 450

layout(vertices = 4) out;

void main() {

  gl_TessLevelOuter[0] = 4;
  gl_TessLevelOuter[1] = 4;
  gl_TessLevelOuter[2] = 4;
  gl_TessLevelOuter[3] = 4;

  gl_TessLevelInner[0] = 4;
  gl_TessLevelInner[1] = 4;

  gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
}
```

Tessellation Shaders : Exemple simple

Etape 3 :

- Le *Tessellation primitive generation* réalise la *tessellation*
- Il découpe le patch en triangle

Mais il travail sur le patch abstrait.

Tessellation Shaders : Exemple simple

Etape 4 :

- Le *tessellation evaluation shader* recoit les coordonnées du *patch* et les coordonnées d'un des sommets issus de la *tessellation* du *patch abstrait*
- Il doit exprimer le sommet issu de la *tessellation* dans le repère correct.

```
#version 450

layout (quads, equal_spacing, ccw) in;

uniform mat4 model_view_matrix;
uniform mat4 projection_matrix;

void main() {

  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);
  vec4 p = mix(p1, p2, gl_TessCoord.y);

  gl_Position = projection_matrix*model_view_matrix*(p,1.0));
}
```

Tessellation Shaders : Exemple simple

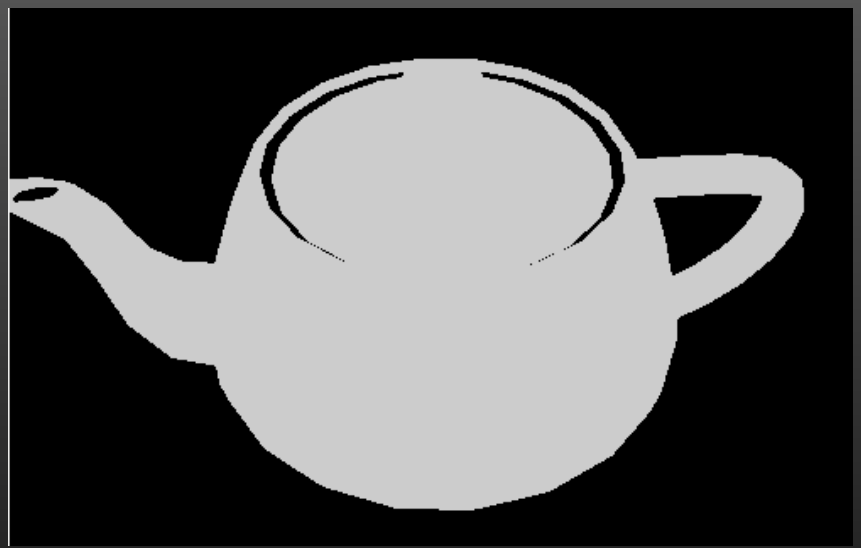
Etape 5 : (sans *geometry shader*)

- Le *fragment shader* reçoit les coordonnées du fragment et le dessine.

```
#version 450

out vec4 output_color;


void main()
{
  output_color = vec4(0.8, 0.8, 0.8, 0.8);
}
```

Tessellation Shaders : Exemple simple

Résultat :



Tessellation Shaders : Exemple 2

Ajout du lissage

Tessellation Shaders : Exemple 2

Dans le *vertex shader* :

```
layout(location = 2) in vec3 vNormal;
(...)

out vec3 vs_normal;
(...)

void main() {
(...)
  vs_normal = vNormal;
(...)
}
```

Tessellation Shaders : Exemple 2

Dans le *tessellation control shader* :

```
in vec3 vs_normal[];
(...)

out vec3 tcs_normal[];
(...)

void main() {
(...)
    tcs_normal[gl_InvocationID] = vs_normal[gl_InvocationID];
(...)
}
```

Tessellation Shaders : Exemple 2

Dans le *tessellation evaluation shader* :

```
in vec3 tcs_normal[];
(...)

out vec3 tes_normal;
(...)

void main() {
(...)
  vec3 n1 = mix(tcs_normal[0], tcs_normal[1], gl_TessCoord.x);
  vec3 n2 = mix(tcs_normal[3], tcs_normal[2], gl_TessCoord.x);
  vec3 n = mix(n1, n2, gl_TessCoord.y);
  tes_normal = n;
(...)
}
```

Tessellation Shaders : Exemple 2

Dans le *fragment shader* :

```
in vec3 tes_normal;
(...)

void main() {
(...)
  output_color = vec4(clamp((dot(tes_normal,...))*color , 0.0, 1.0), 1.0);
(...)
}
```

Tessellation Shaders : Exemple 2

Résultat :

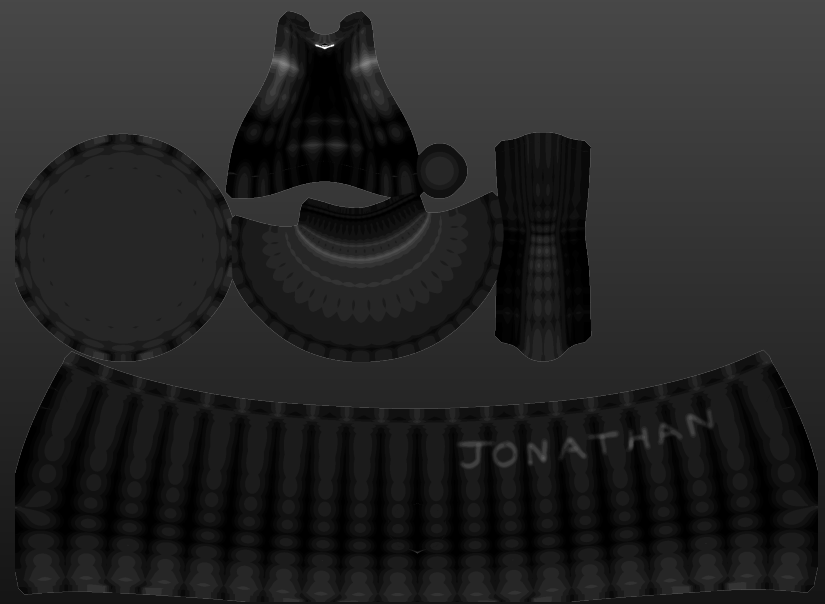
Tessellation Shaders : exemple de *height map*

Changement de position des sommets.

Tessellation Shaders : exemple de *height map*

Changement de position des sommets.
Pour cela, il nous faut :

- une texture avec les hauteurs
- ajouter les coordonnées textures *uv* des sommets (comme on a fait pour les normales)



Tessellation Shaders : exemple de *height map*

Changement de position des sommets

Pour rappel dans le *tessellation control shader* :

```
(...)
  gl_TessLevelOuter[0] = 4;
  gl_TessLevelOuter[1] = 4;
  gl_TessLevelOuter[2] = 4;
  gl_TessLevelOuter[3] = 4;

  gl_TessLevelInner[0] = 4;
  gl_TessLevelInner[1] = 4;
(...)
```

Tessellation Shaders : exemple de *height map*

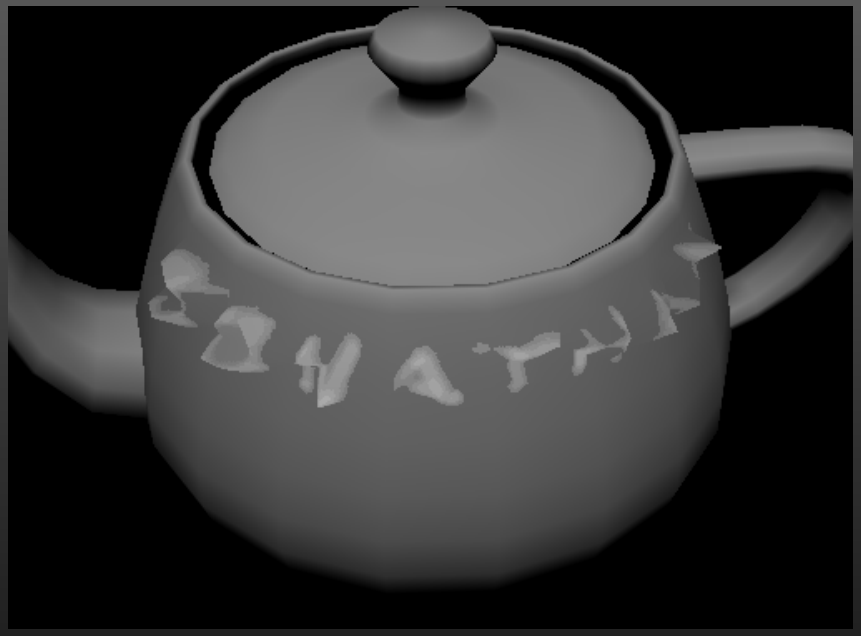
Changement de position des sommets

Dans le *tessellation evaluation shader* :

```
(...)
uniform sampler2D heightmap_sampler;
(...)
float k = clamp((texture(heightmap_sampler, uv)).x, 0.0, 1.0);
(...)
void main() {
(...)
gl_Position = projection_matrix*model_view_matrix*(p+vec4(k*n,1.0));
(...)
}
```

Tessellation Shaders : exemple de *height map*

Résultat :



Tessellation Shaders : exemple de *height map*

Changement de position des sommets

Changement dans le *tessellation control shader* :

```
(...)
  gl_TessLevelOuter[0] = 50;
  gl_TessLevelOuter[1] = 50;
  gl_TessLevelOuter[2] = 50;
  gl_TessLevelOuter[3] = 50;

  gl_TessLevelInner[0] = 50;
  gl_TessLevelInner[1] = 50;
(...)
```

Tessellation Shaders : exemple de *height map*

Résultat :

Tessellation Shaders : exemple de *height map*

Résultat :

Tessellation Shaders : exemple de *height map*

Note la carte des hauteurs est imparfaite :

Ce qui explique certains artefacts.

Tessellation Shaders : Retour sur l'input

On peut envoyer des *patches* autres que des quatrilaters triangle ou ligne.

- Le *draw* est invoqué avec la primitive : *GL_PATCHES*
- On peut contrôler la taille du *patch* avec *glPatchParameteri*(*GL_PATCH_VERTICES*, *GLint value*);

On peut donc passer plus de sommets comme par exemple le voisinage... C'est pour cela que l'on peut changer la quantité de éléments par *patch*.

Tessellation Shaders : conclusion

- Permet d'enrichir le model.
- Bien refléchir à chaque fois si cela relève du *geometry shader* ou du *tessellation shader*.

Conclusion

Ces *shaders* permettent de deporter au maximum les calculs sur GPUs. Les *Geometry Shader* et *Tessellation Shader* feront le travail pendant le rendu, le *Compute Shader* le fera en dehors du rendu.