M2-Images

Rendu Temps Réel - OpenGL 4 et tessellation

J.C. Iehl

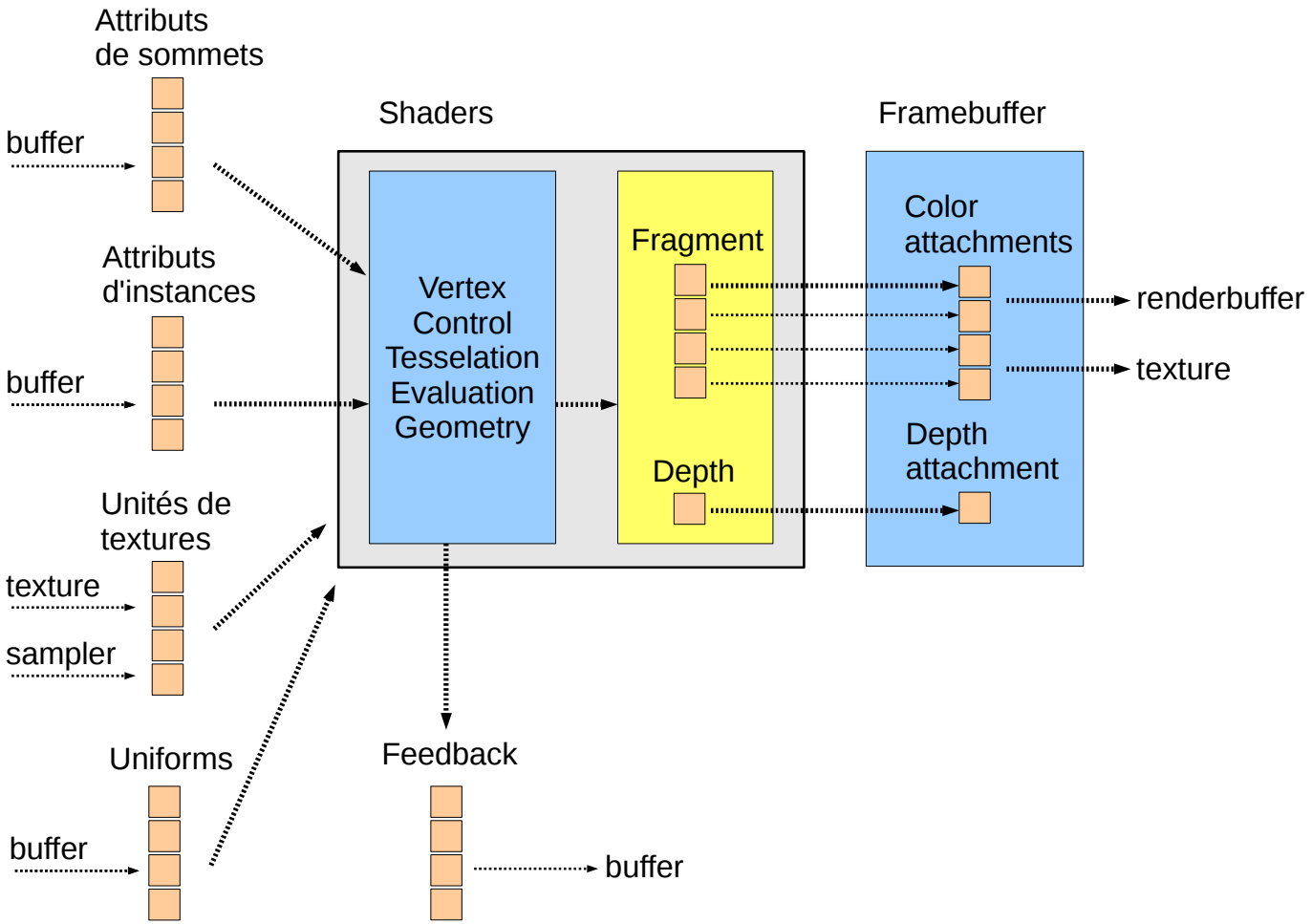
April 12, 2012

Résumé des épisodes précédents

résumé :

- création de buffers,
- création de maillages indexés ou non,
- affichage de maillages,
- affichage de plusieurs maillages,
- vertex, géométrie et fragment shaders,
- textures, framebuffer,
- notions de traitement en plusieurs passes ...

Résumé de l'api openGL 4



Différences principales entre GL3 et GL4

plusieurs catégories :

- performances : API plus efficace,
- fonctionnalités : nouvelles étapes programmables.

openGL4 : nouvelles étapes

Subdivision à la volée / Tessellation :

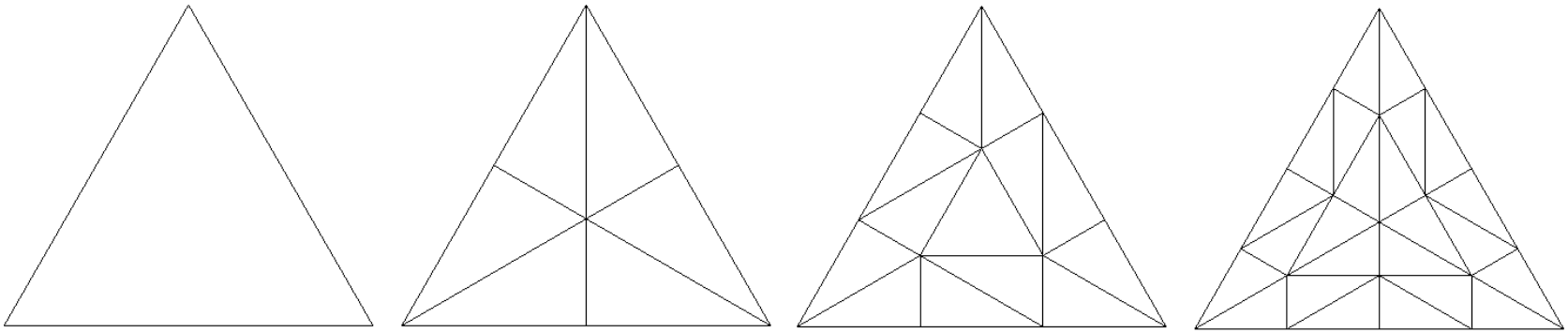
- un nouveau type de primitive : `GL_PATCH`,
- nombre de *points de contrôle* quelconque (< 32),
- mais chaque patch doit être subdivisé pour être dessiné,
- subdivision en triangles,
- chaque triangle est ensuite envoyé au reste du pipeline,
- ... cf REYES ?

afficher un carreau de Bézier, par exemple, ou un triangle de Bézier, directement...

openGL4 : Subdivision

- 3 nouvelles étapes dans le pipeline pour manipuler les patchs :
- Control shader,
- Tessellation unit,
- Evaluation shader.

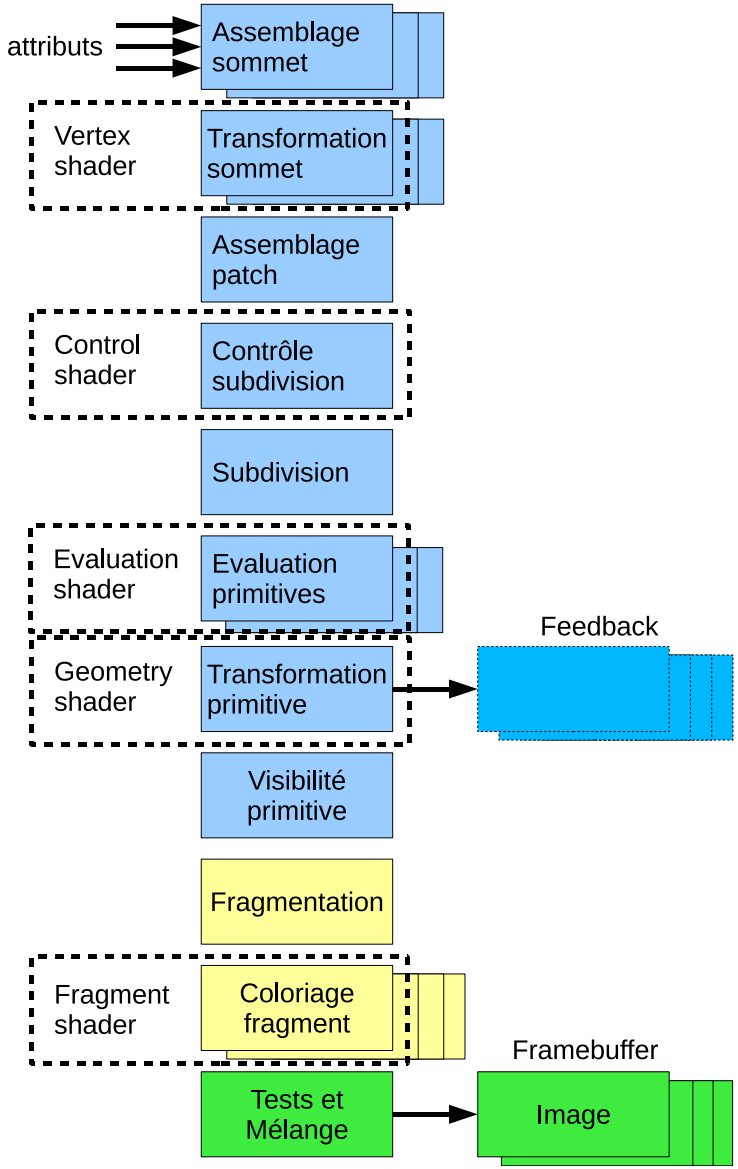
l'étape tessellation est configurable mais pas programmable.



openGL4 : pipeline

pipeline complet :

- vertex shader,
- control shader,
- tessellation unit,
- evaluation shader,
- geometry shader,
- assemblage des primitives, élimination des primitives cachées,
- fragment shader,
- tests et écriture dans le framebuffer.


attributs
Assemblage sommet
Vertex shader
Transformation sommet
Assemblage patch
Control shader
Contrôle subdivision
Subdivision
Evaluation shader
Evaluation primitives
Feedback
Geometry shader
Transformation primitive
Visibilité primitive
Fragmentation
Fragment shader
Coloriage fragment
Framebuffer
Tests et Mélange
Image

Vertex shader

- pas de modifications particulières,
- exécuté une fois par sommet des patchs dessinés par l'application.

Vertex Shader

Control Shader

nouveau shader :

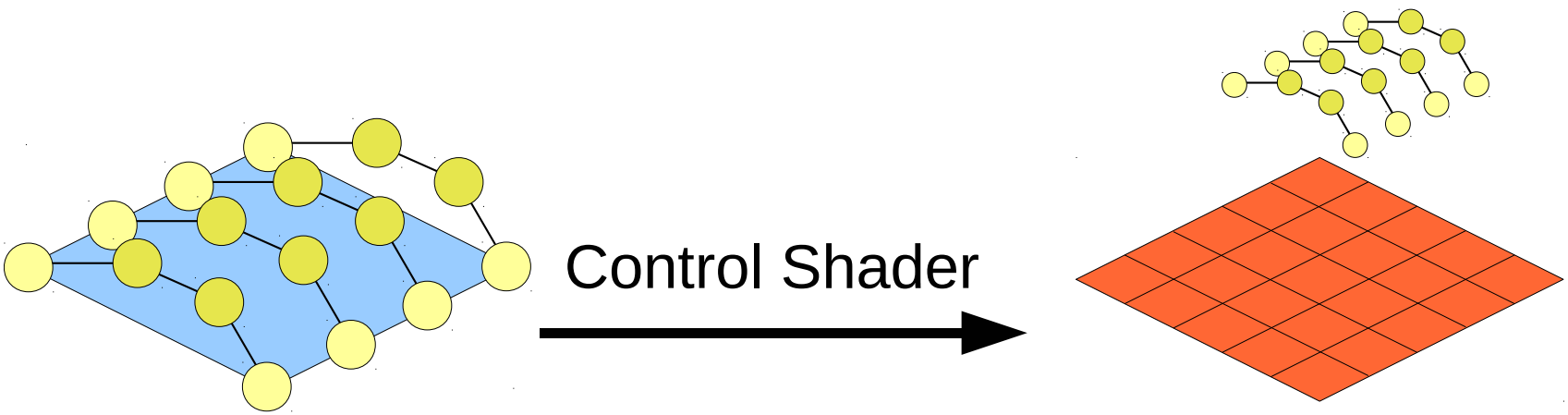
- traite un patch complet
  (ensemble de points de contrôle + attributs),
- doit produire une primitive *"support"* (quad ou triangle), qui sera subdivisée par l'unité de tessellation,
- doit paramètrer la subdivision de la primitive support.

Control Shader

subdiviser la primitive support :

- découpage réalisé dans le *"domaine paramétrique"* du patch,
- paramètre l'unité de tessellation : nombre de découpage par dimension du domaine paramétrique,
- 2 pour les quads, 3 pour les triangles,
- pour l'intérieur du patch...
- ... et les arêtes.

Control Shader



Control Shader

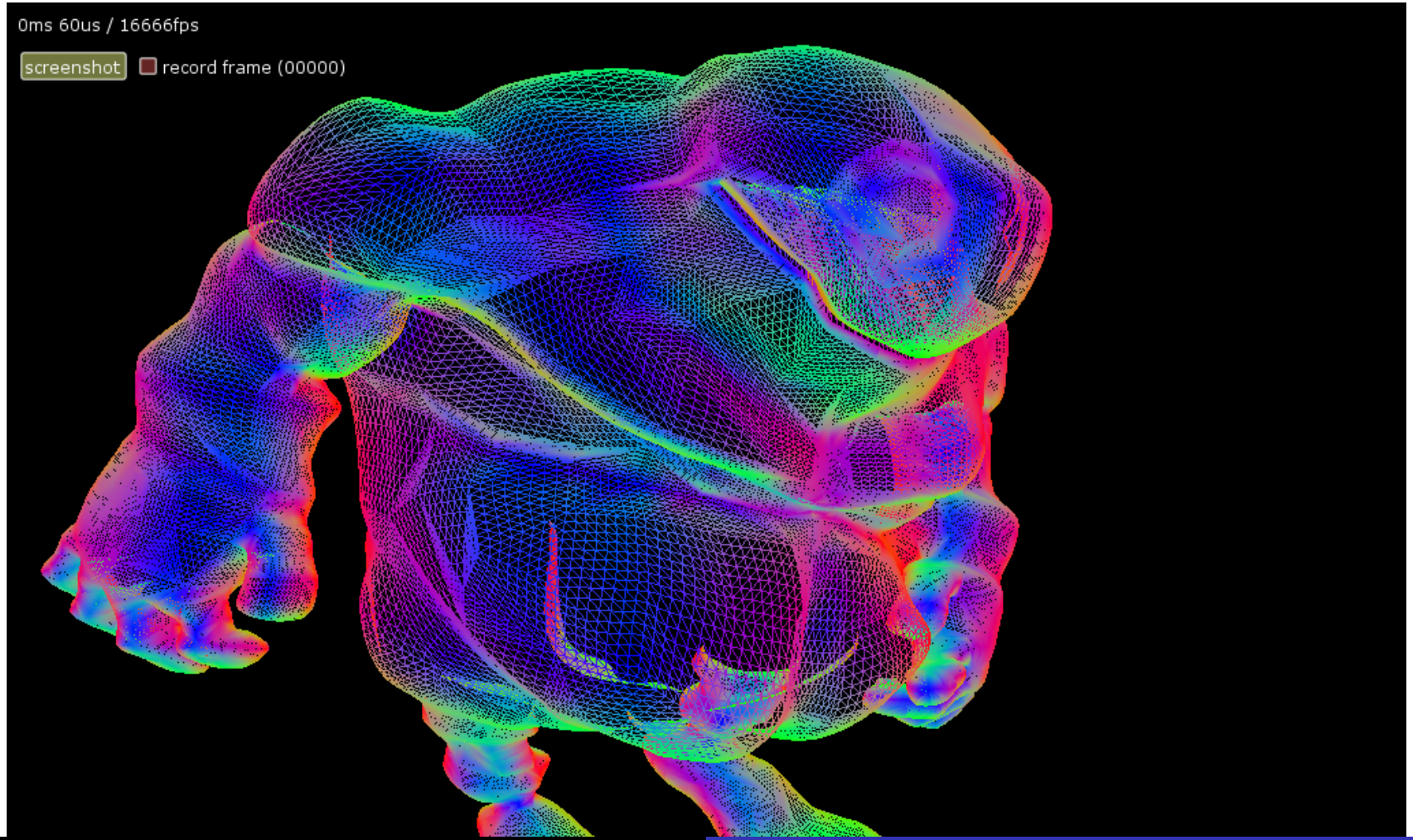
remarque :

- nombre de découpes *réel*, on peut découper 2.65 fois une arête...
- pourquoi par arete ? pourquoi paramétrer l'interieur ?

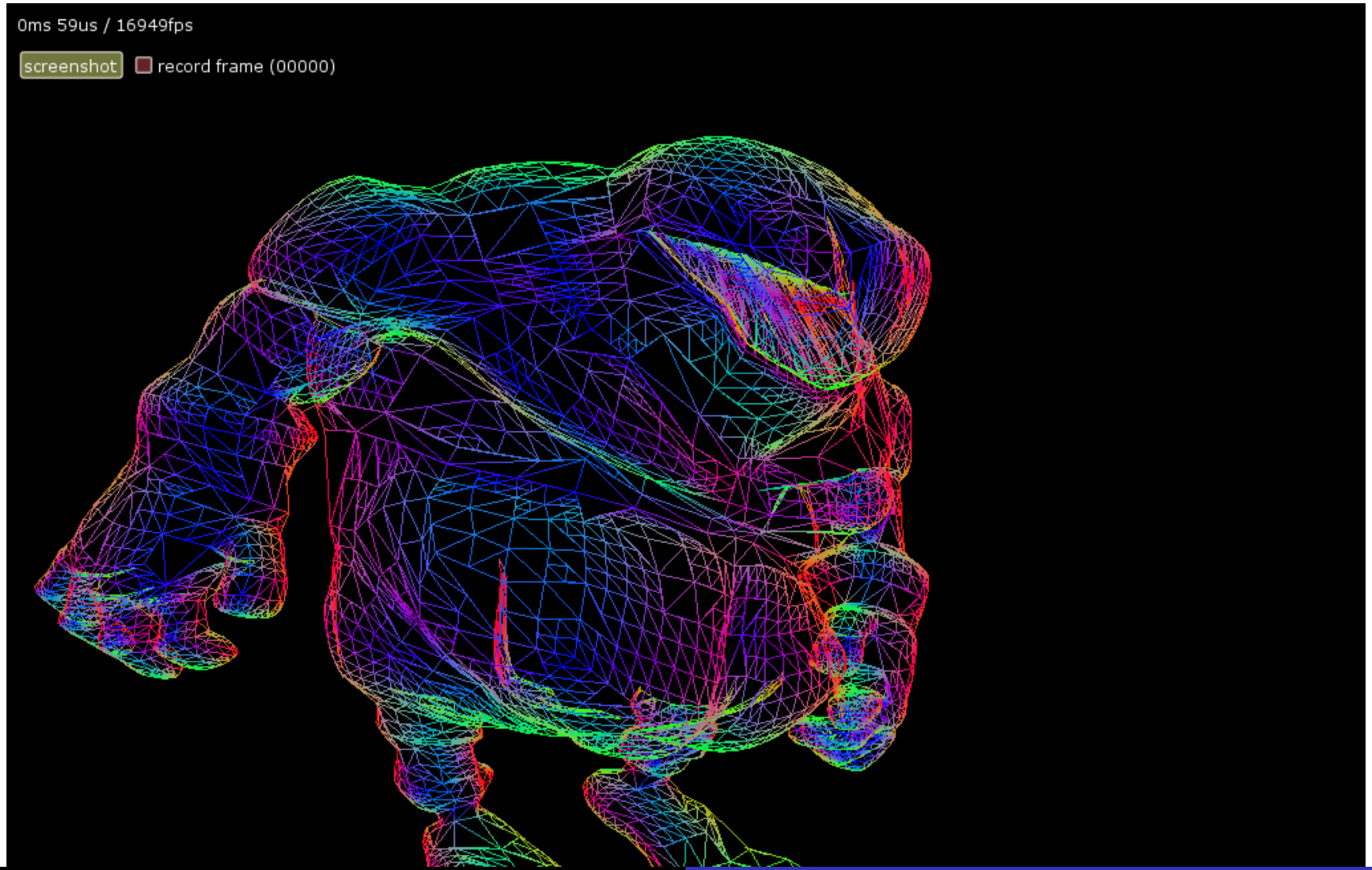
demo

Subdivision régulière: exemple

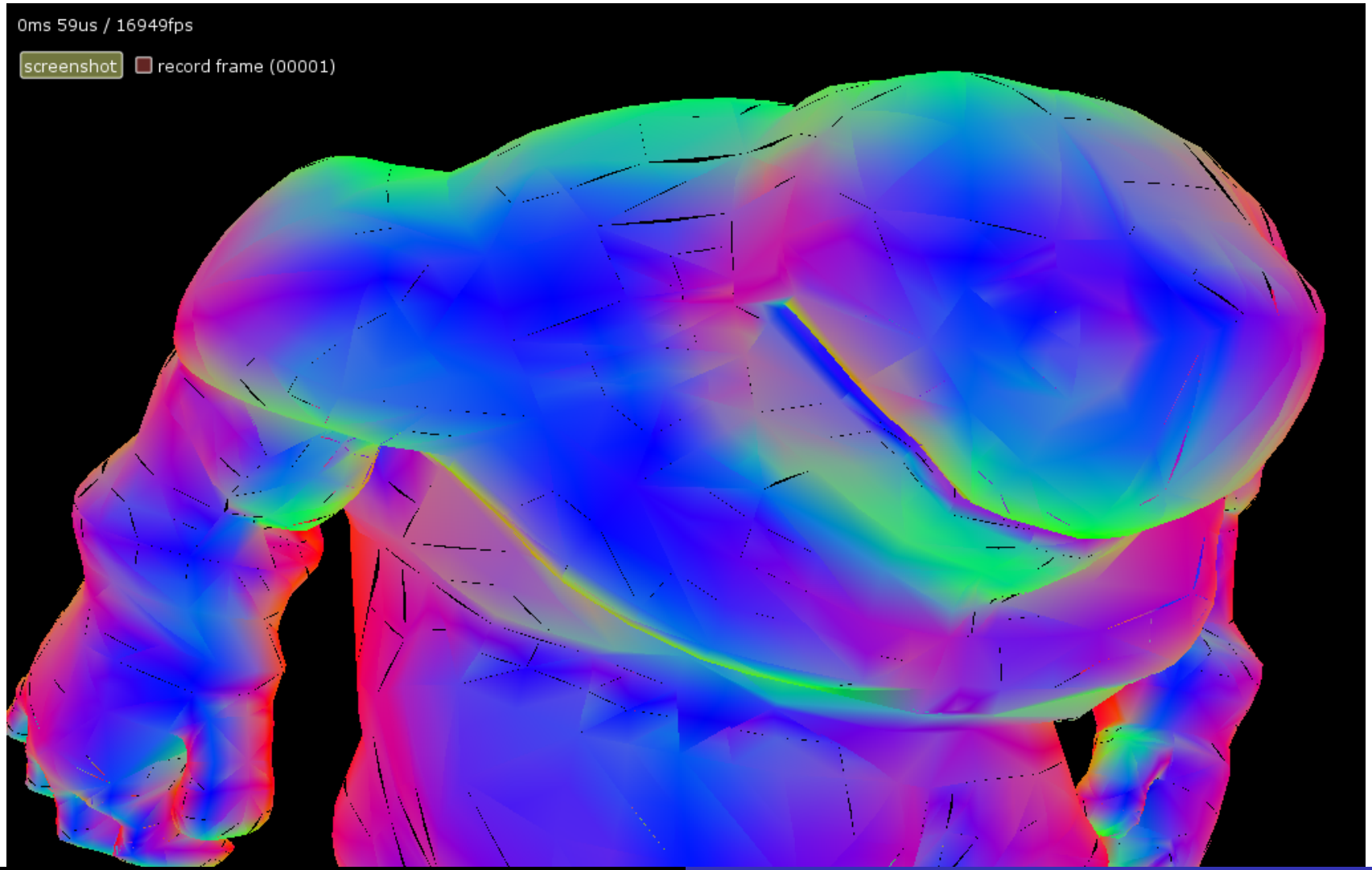
PN Triangles / Triangles de Beziers



Subdivision adaptive: problèmes de raccords



Subdivision adaptative : problèmes de raccords



Control shader : contrôle de la subdivision

évaluer le critère de subdivision :

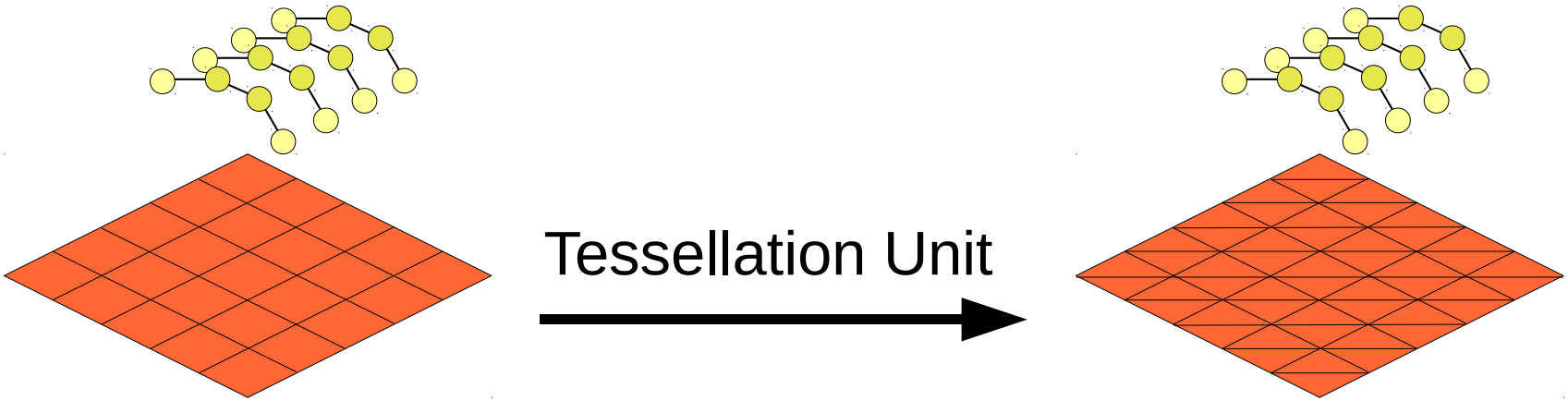
- par arête, pour éviter les "trous" dans le maillage généré.

Unité de tessellation

nouvelle unité paramétrable :

- produit un maillage triangulé de la primitive support émise par le control shader,
- (domaine paramétrique du patch,)
- en fonction des paramètres indiqués par le control shader,
- (subdivision d'un carre / d'un triangle),
- ... et ceux déclarés par l'evaluation shader.

Tessellation Unit



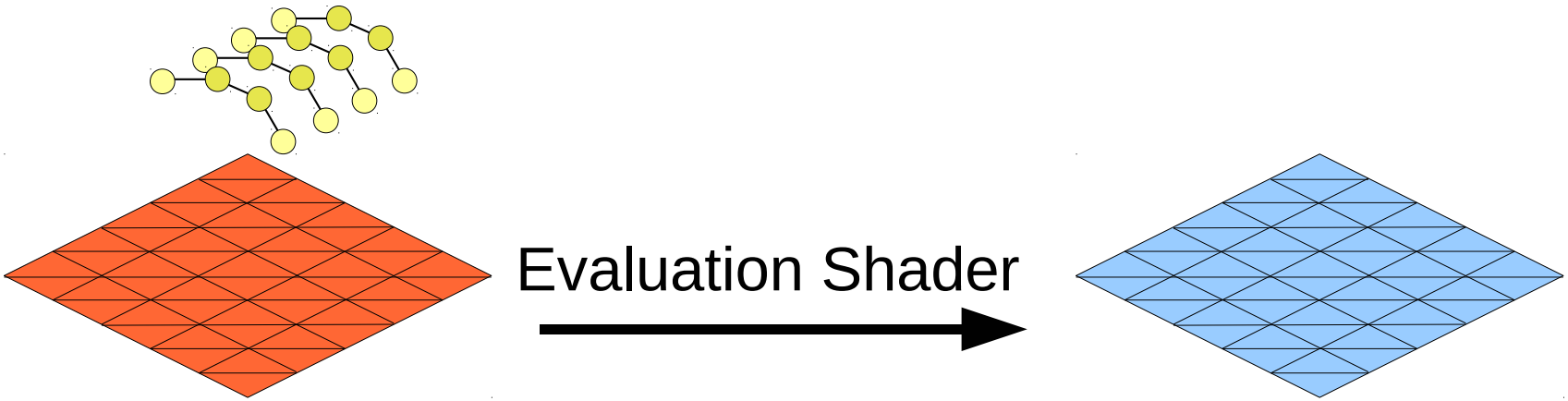
Evaluation shader

nouveau shader :

- paramètre le maillage produit par l'unité de tessellation,
- traite les sommets du maillage,
- doit produire les sommets dans le repère projectif homogène de la caméra,
- (accès aux attributs de chaque sommet du patch découpé),

passage entre le *domaine paramétrique* du patch et le repère projectif de la caméra ?

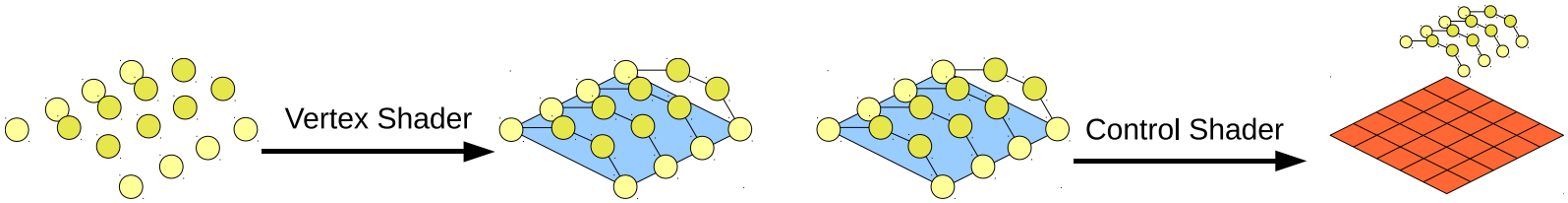
Evaluation Shader

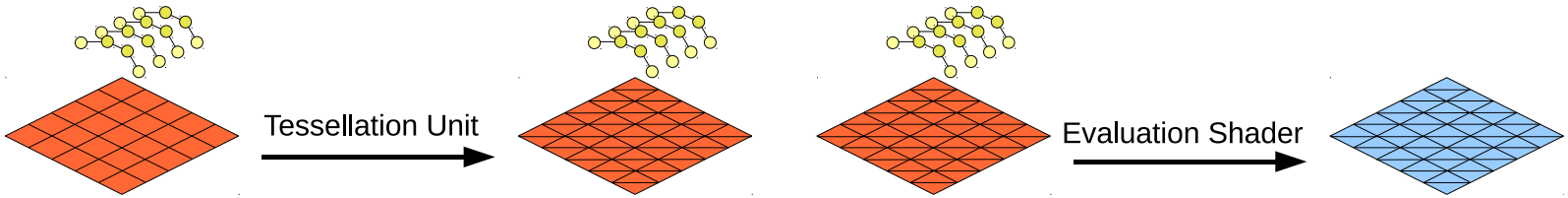


Geometry shader

- pas de modifications particulières,
- exécuté une fois par triangle produit par l'evaluation shader,
- doit produire les sommets dans le repère projectif de la camera.

Résumé du pipeline





Control shader en détails...

en entrée :

- positions des points de contrôle : `gl_in[].gl_Position`,

en sortie :

- déclarer le nombre de sommets de la primitive support : `layout(vertices= n) out;`
- positions des sommets : `gl_out[].gl_Position`,
- nombre de subdivisions : `gl_TessLevelInner[]`, `gl_TessLevelOuter[]`,

Control shader en détails....

en sortie :

- attributs supplémentaires, texture, etc.

remarques :

- subdivision d'un triangle (primitive support) :
- `patch out float gl_TessLevelInner[1]`,
- `patch out float gl_TessLevelOuter[3]`,
- subdivision d'un quad (primitive support) :
- `patch out float gl_TessLevelInner[2]`,
- `patch out float gl_TessLevelOuter[4]`.

Control shader en détails...

nouveau type d'attribut (par patch):

`patch out/in`

valeurs communes à tous les sommets du patch...

le shader est exécuté une fois par sommet :

- une fois par sommet à produire,
- mais peut accéder à l'ensemble des entrées `gl_in[]`,
- ne produire qu'un seul sommet de la primitive support ?
- `gl_InvocationID`, indice du sommet traité.

```
// control shader pour mailler des triangles
#version 410

// declare le nombre de sommet a produire, 3 pour un triangle
layout(vertices= 3) out;

uniform float inner_factor;
uniform vec3 edge_factor;

void main( void )
{
    // copie la position du sommet pour l'etape suivante du pipeline
    // gl_InvocationID est l'index du sommet a traiter
    gl_out[gl_InvocationID].gl_Position=
        gl_in[gl_InvocationID].gl_Position;

    // parametre l'unite de tessellation
    gl_TessLevelInner[0]= inner_factor;

    gl_TessLevelOuter[0]= edge_factor.x;
    gl_TessLevelOuter[1]= edge_factor.y;
    gl_TessLevelOuter[2]= edge_factor.z;
}
```

```
// control shader pour mailler des quads
#version 410

// declare le nombre de sommet a produire, 4 pour un quad
layout(vertices= 4) out;

uniform vec2 inner_factor;
uniform vec4 edge_factor;

void main( void )
{
    // copie la position du sommet pour l'etape suivante du pipeline
    // gl_InvocationID est l'index du sommet a traiter
    gl_out[gl_InvocationID].gl_Position=
        gl_in[gl_InvocationID].gl_Position;

    // parametre l'unite de tessellation
    gl_TessLevelInner[0]= inner_factor.x;
    gl_TessLevelInner[1]= inner_factor.y;

    gl_TessLevelOuter[0]= edge_factor.x;
    gl_TessLevelOuter[1]= edge_factor.y;
    gl_TessLevelOuter[2]= edge_factor.z;
    gl_TessLevelOuter[3]= edge_factor.w;
}
```

Evaluation Shader en détails...

en entrée :

- type de primitive support / type de découpage :
- `layout(triangles) in;` un triangle subdivisé en triangles,
- `layout(quads) in;` un quad subdivisé en triangles,
- méthode d'arrondi du nombre de subdivisions calculé par le control shader :
- `layout(equal_spacing) in;`
- `layout(fractional_even_spacing) in;`
- `layout(fractional_odd_spacing) in;`

Evaluation Shader en détails...

en entrée : triangles résultat de la subdivision

- `gl_in[].gl_Position` des sommets de la primitive support,
- + les attributs déclarés en sortie du control shader,
- `gl_TessCoord` : position paramétrique du sommet (dans la primitive support),
- `gl_TessCoord.xy` pour un quad,
- `gl_TessCoord.xyz` pour un triangle.

en sortie :

- `gl_Position` du sommet traité (à calculer en fonction de `gl_TessCoord`).

```
// evaluation shader pour mailler des triangles
#version 410

uniform mat4 mvpMatrix;

// parametre l'unite de decoupage
layout(triangles, fractional_odd_spacing, ccw) in;

void main( void )
{
    // recupere la position dans le domaine parametrique
    // du sommet a traiter pour cette execution du shader
    float w= gl_TessCoord.z;
    float u= gl_TessCoord.x;
    float v= gl_TessCoord.y;

    // calcule la position du sommet dans le repere local du
    // triangle abc, interpolation barycentrique
    // p(u, v, w)= a*w + b*u + c*v,
    vec3 position= gl_in[0].gl_Position.xyz * w
        + gl_in[1].gl_Position.xyz * u
        + gl_in[2].gl_Position.xyz * v;

    // transforme le point dans le repere projectif
    gl_Position= mvpMatrix * vec4(position, 1.f);
}
```

```
// evaluation shader pour mailler des quads
#version 410

uniform mat4 mvpMatrix;

// parametre l'unite de decoupage
layout(quads, fractional_odd_spacing, ccw) in;

void main( void )
{
    // recupere la position dans le domaine parametrique
    // du sommet a traiter pour cette execution du shader
    float u= gl_TessCoord.x;
    float v= gl_TessCoord.y;

    // calcule la position du sommet dans le repere local du quad abcd,
    // interpolation barycentrique p(u, v)= a + (b-a)*u + (d-a)*v;
    // l'arete ab est utilisee comme axe u, et l'arete ad pour l'axe v.
    vec3 position= gl_in[0].gl_Position.xyz
        + u * (gl_in[1].gl_Position.xyz - gl_in[0].gl_Position.xyz)
        + v * (gl_in[3].gl_Position.xyz - gl_in[0].gl_Position.xyz);

    // transforme le point dans le repere projectif
    gl_Position= mvpMatrix * vec4(position, 1.f);
}
```

Evaluation shader en détail...

en sortie :

- `gl_Position` du sommet traité
  (à calculer en fonction de `gl_TessCoord`).
- cette information peut être calculée avant (vertex, control)...
- ... ou après (geometry shader).

Tessellation en détails...

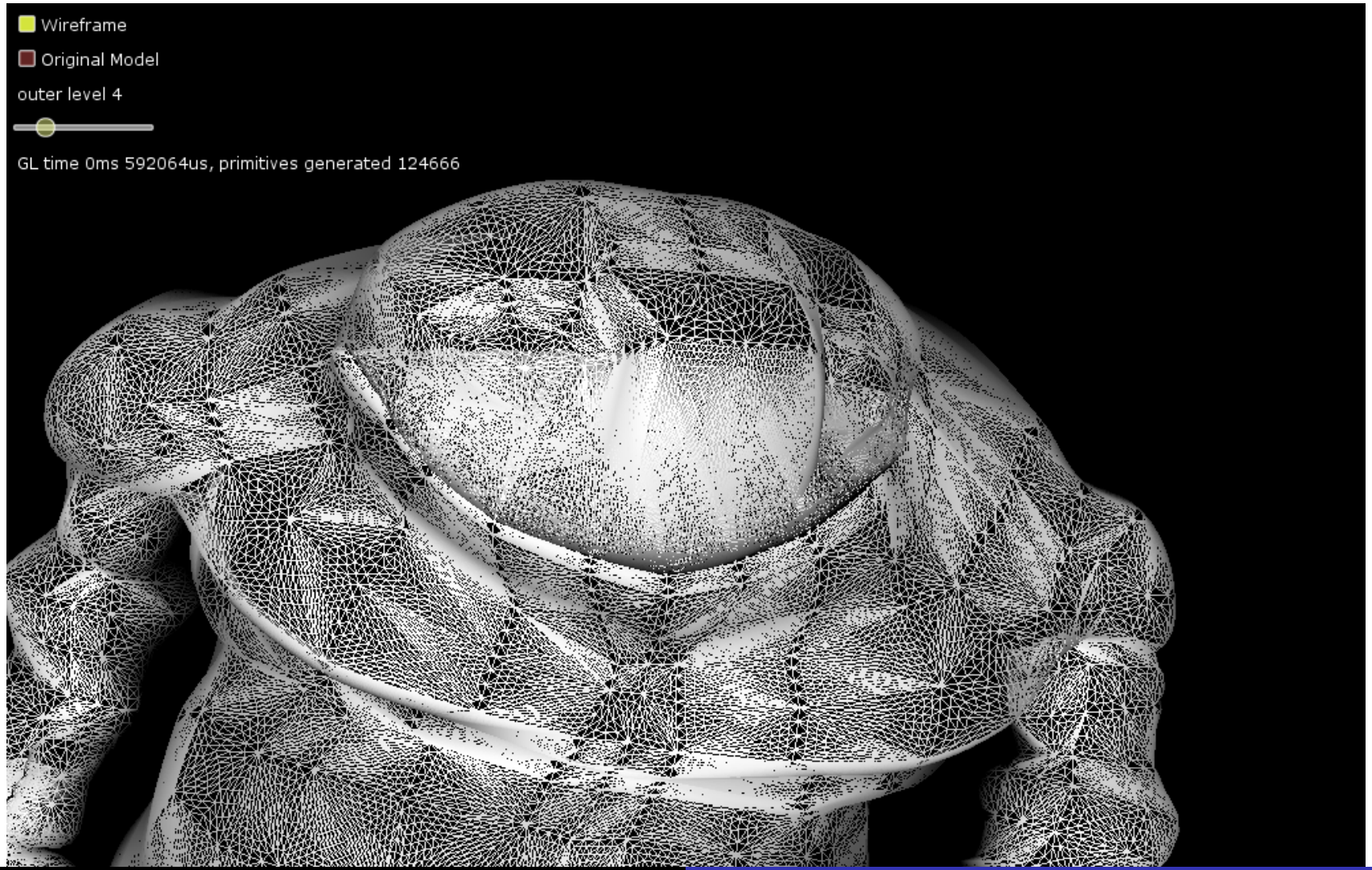
- spécification openGL4 : http://www.opengl.org/registry/
- GLSL, "Builtin variables", chapitre 7.

Tessellation efficace ?

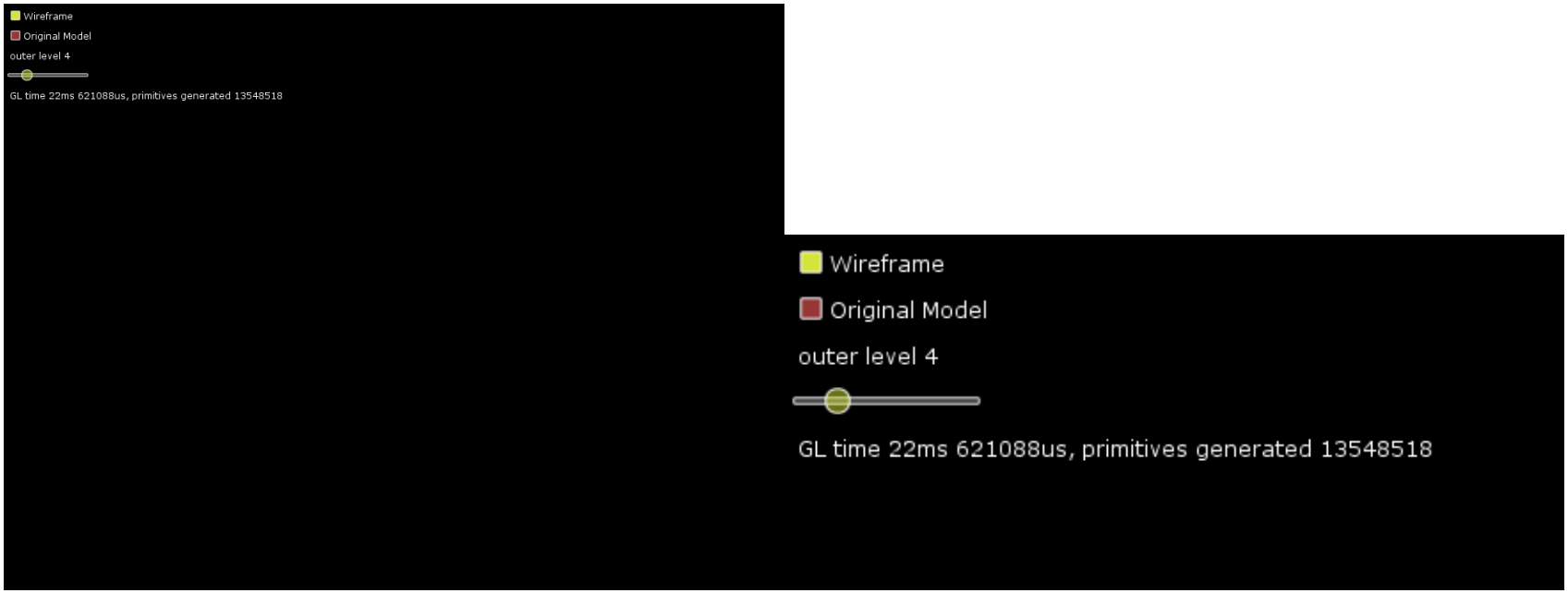
oui, mais...

- subdivision adaptative :
- en fonction de la longueur des arêtes dans l'image,
- en fonction de la courbure de la surface,
- sur la silhouette,
- la subdivision génère beaucoup de triangles,
- ils seront éliminés / clippés *après* le geometry shader,
- subdivision d'un patch sur le plan near ?

Tessellation efficace ?



Tessellation efficace ?



aucun patch n'est visible... mais la tessellation à produit >13M triangles...

Tessellation efficace ?

comment limiter le nombre de triangles produits ?

- ne pas subdiviser lorsque c'est inutile !
- `gl_TessLevelInner[]= 0.f;`
- `gl_TessLevelOuter[]= 0.f;`

remarque : utiliser une requête `GL_PRIMITIVE_GENERATED` pour compter les triangles produits par la subdivision.

compromis habituel : plus de souplesse mais moins d'automatismes... problème équivalent avec l'instanciation.

Exécution parallèle

- inutile de remplir les paramètres de subdivision à chaque exécution du shader
  (déclaration `patch out float gl_TessLevelOuter[4]`),
- inutile de remplir chaque paramètre patch out à chaque exécution du shader,

Exécution parallèle

```
// control shader pour mailler des triangles
...

void main( void )
{
    // copie la position du sommet pour l'etape suivante du pipeline
    // gl_InvocationID est l'index du sommet a traiter
    gl_out[gl_InvocationID].gl_Position=
        gl_in[gl_InvocationID].gl_Position;

    if(gl_InvocationID == 0)
    {
        // parametre l'unite de tessellation,
        // lorsque le sommet 0 est traite
        gl_TessLevelInner[0]= inner_factor;

        gl_TessLevelOuter[0]= edge_factor.x;
        gl_TessLevelOuter[1]= edge_factor.y;
        gl_TessLevelOuter[2]= edge_factor.z;
    }
}
```

Exécution parallèle

- tous les shaders d'une étape du pipeline s'exécutent en même temps...
- tous les vertex shaders des points de contrôle d'un ou plusieurs patchs,
- tous les control shaders, tous les evaluation shaders,
- répartir le 'travail' entre les exécution des shaders.

Exécution parallèle

- séparer le shader en 2 :
- la première partie réalise un traitement 'local'
  (sur chaque sommet / arête),
- la deuxième peut réaliser un traitement 'global'
  (sur le patch, utilise les infos calculés par sommet)
- utiliser `barrier()`
  pour synchroniser l'exécution entre les 2 parties.

Exécution parallèle

exemple : calcul par arete + calcul par patch

- calculer le niveau de subdivision de l'arete i :
- entre les sommets i et $(i+1)\%3$,
- calculer la longueur de l'arete en pixels (projettée sur l'écran),
- en déduire le nombre de subdivisions pour construire des aretes de n pixels,

Exécution parallèle

idée :

- chaque shader projette le sommet i,
- `barrier();`
- finir le calcul de longueur d'arete et de nombre de subdivision par arete,
- quelle subdivision pour l'intérieur du patch ?

résultat :

- évite de recalculer plusieurs fois les mêmes transformations...

Exécution parallèle

générer beaucoup de petits triangles :

- est *très* inefficace...
- la fragmentation des primitives est calibrée pour des triangles de ≈8 pixels...
- la parallélisation des taches n'est plus régulière...

"Scheduling the graphics pipeline"
J. Ragan-Kelley, 2011

Fragmentation de petits triangles

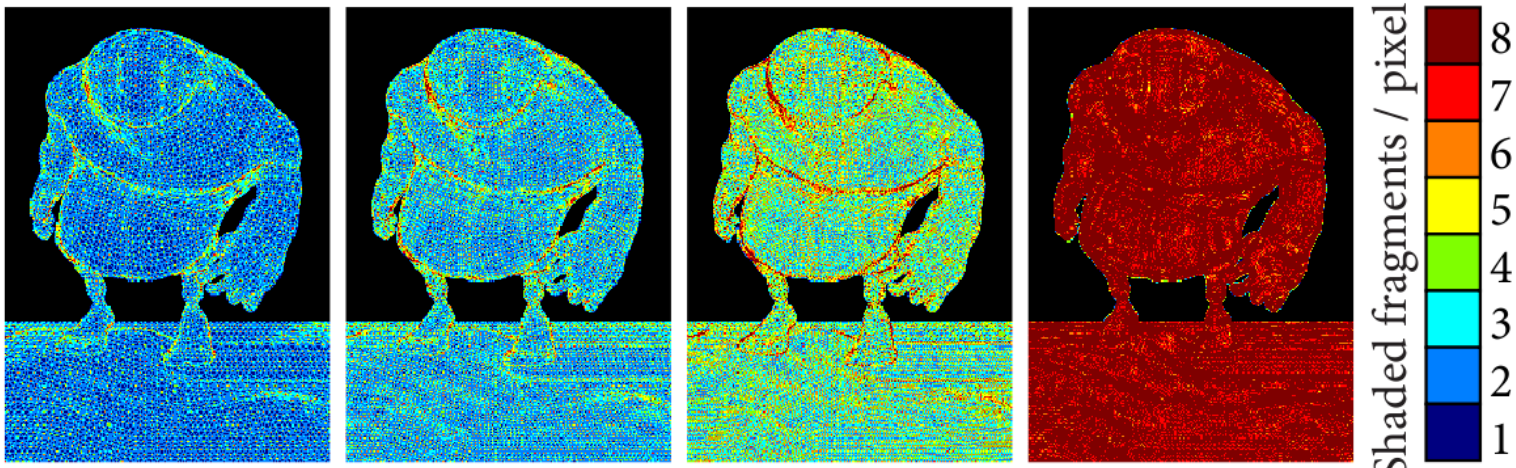
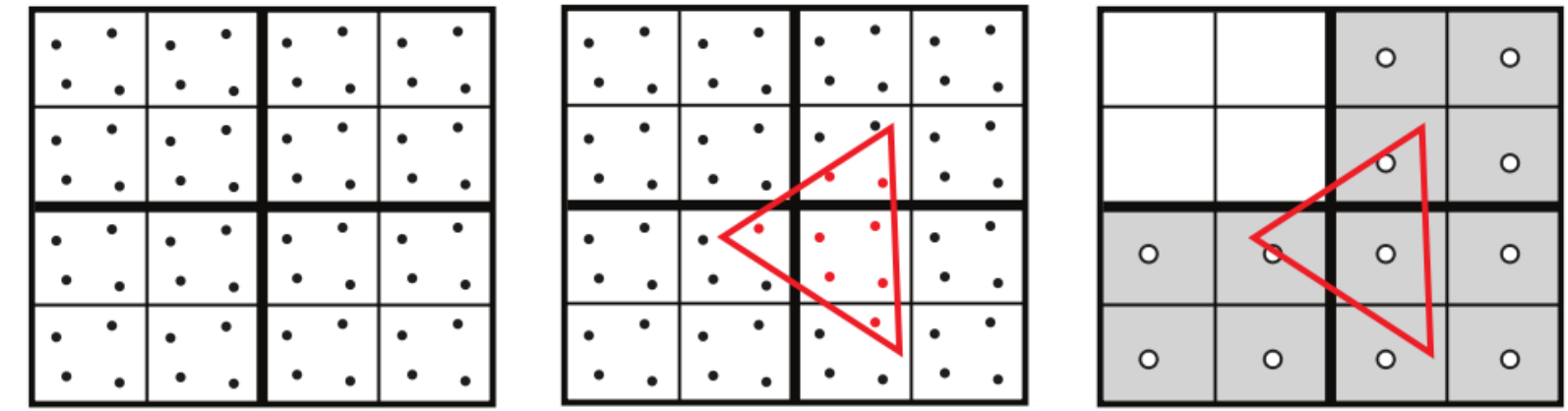


Figure 1: *Over-shade in a GPU pipeline increases as scene triangle size shrinks (images are colored according to the number of fragments shaded per pixel). When this scene is tessellated into pixel-sized triangles, each pixel is shaded more than eight times.*

"Reducing Shading on GPUs using Quad-Fragment Merging" 2010

Fragmentation de petits triangles



- les triangles sont fragmentés par bloc,
- les pixels de chaque bloc contenant le triangle sont testés en parallèle,
- la taille du bloc est fixe...
- plus les triangles sont petits, plus les tests d'inclusion sont inefficaces.

Fragmentation de petits triangles

ne pas générer de petits triangles :

- meilleurs critères de subdivision ?
- ??

"Efficient Pixel-Accurate Rendering of Curved Surfaces"
Y.I Yeo, L. Bin, J. Peters, i3d 2012