Visualisation de données – Programmation OpenGL

Sébastien Limet, Emmanuel Melin et Sophie Robert

Université d'Orléans

2017-2018

Plan

Visualisation scientifique: Qu'est que c'est? À quoi ça sert?

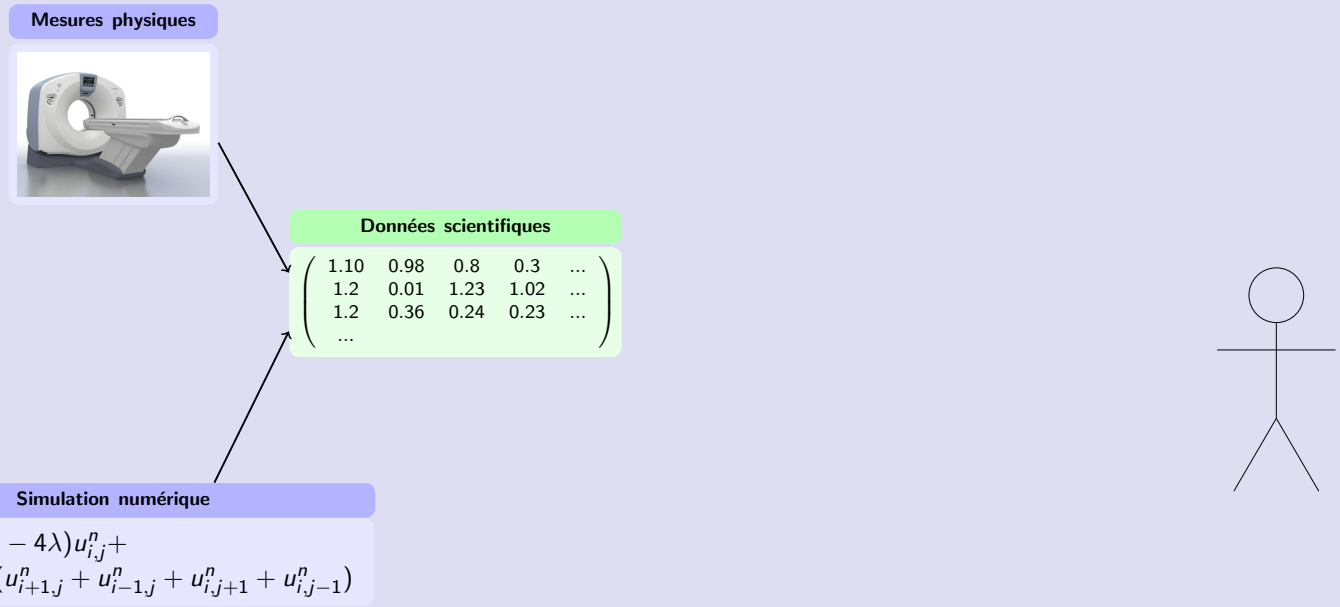
- Utiliser des images (2D ou 3D) à partir de données
  - issues de mesures (scanner, capteurs etc...)
  - calculées par simulation numérique

  Note à l'origine les images ne sont pas nécessairement produites par ordinateur.
- Synthétiser des données complexes sous forme d'images pour aider le scientifique à mieux les comprendre.
- Peu importe la manière d'obtenir les images, l'important est l'image obtenue elle-même.

Visualisation scientifique: Qu'est que c'est? À quoi ça sert?

- De nombreuses techniques permettent d'obtenir des images à partir de données scientifiques
- VTK est un outil permettant d'intégrer beaucoup de ces techniques pour obtenir des images pertinentes
- Les techniques utilisées et les images souhaitées dépendent fortement
  - des données à visualiser
  - du domaine scientifique

Boucle d'interaction visuelle



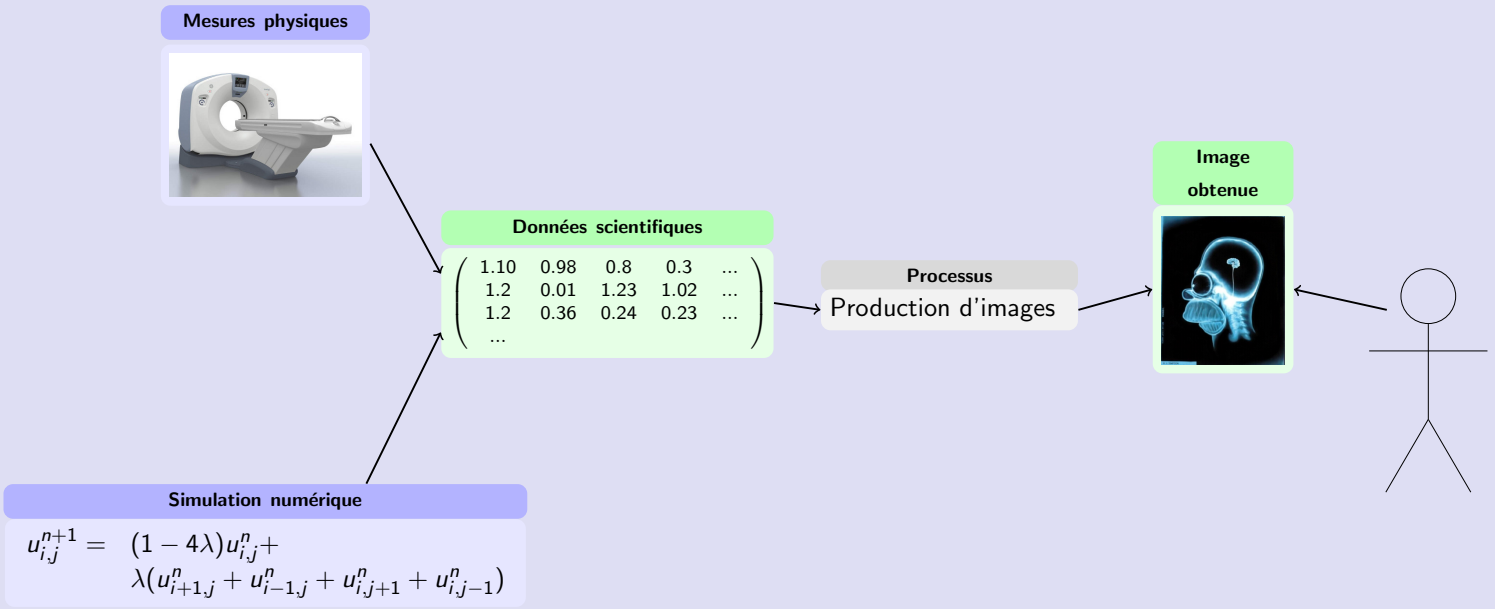
Des données sont produites

- soit par des mesures,
- soit par simulation

Boucle d'interaction visuelle

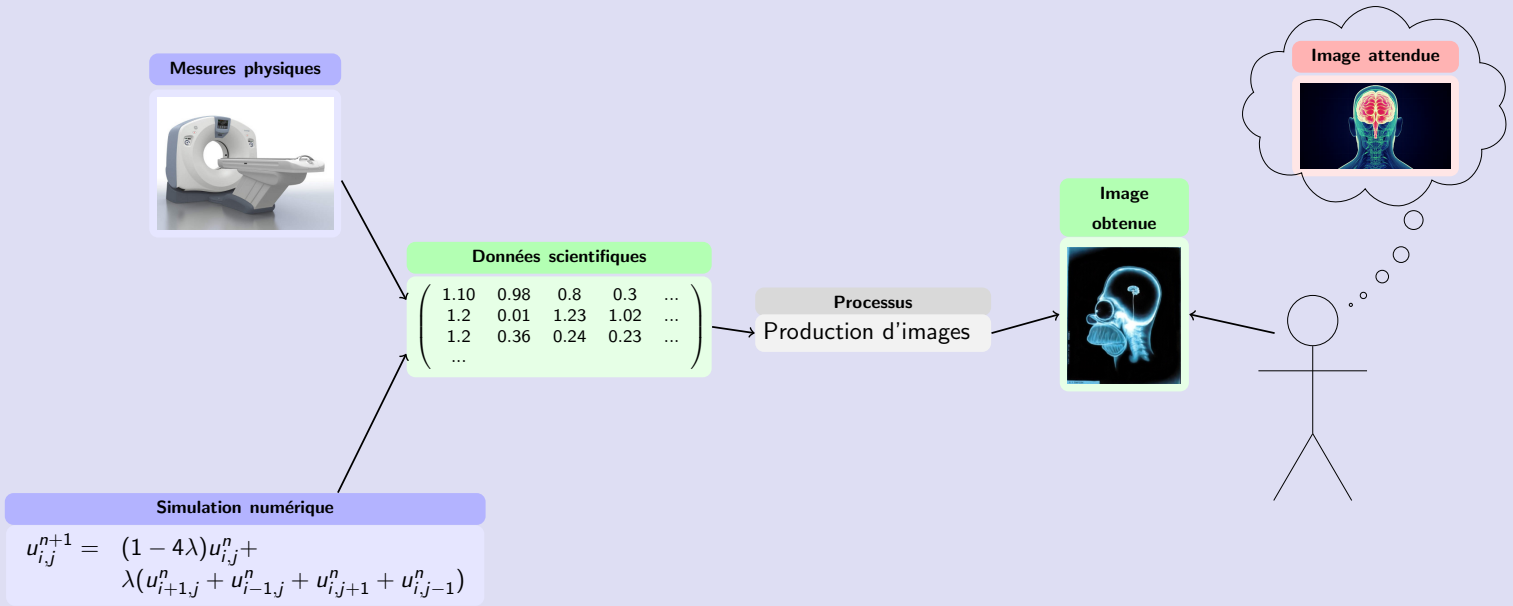
Un processus de traitement permet de produire des images

Boucle d'interaction visuelle



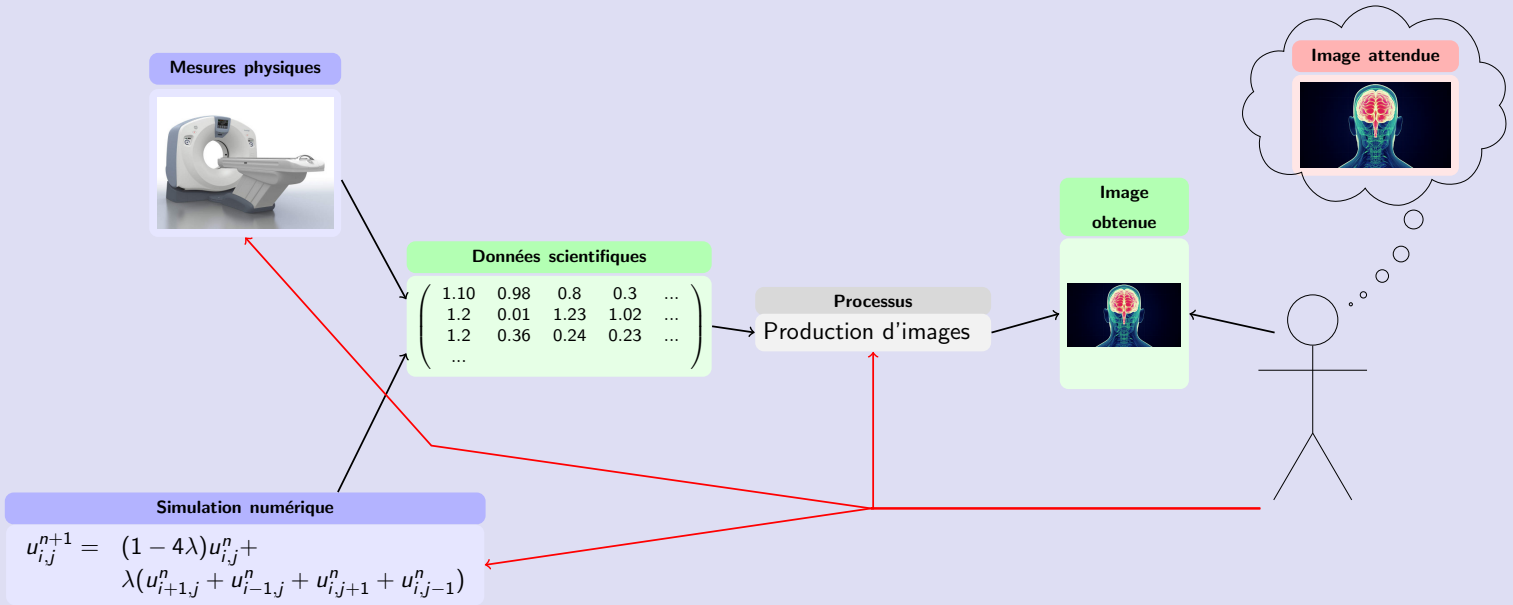
Le scientifique vérifie que l'image produite semble cohérente

Boucle d'interaction visuelle



Le scientifique vérifie que l'image produite semble cohérente avec ce qu'il s'attendait à obtenir

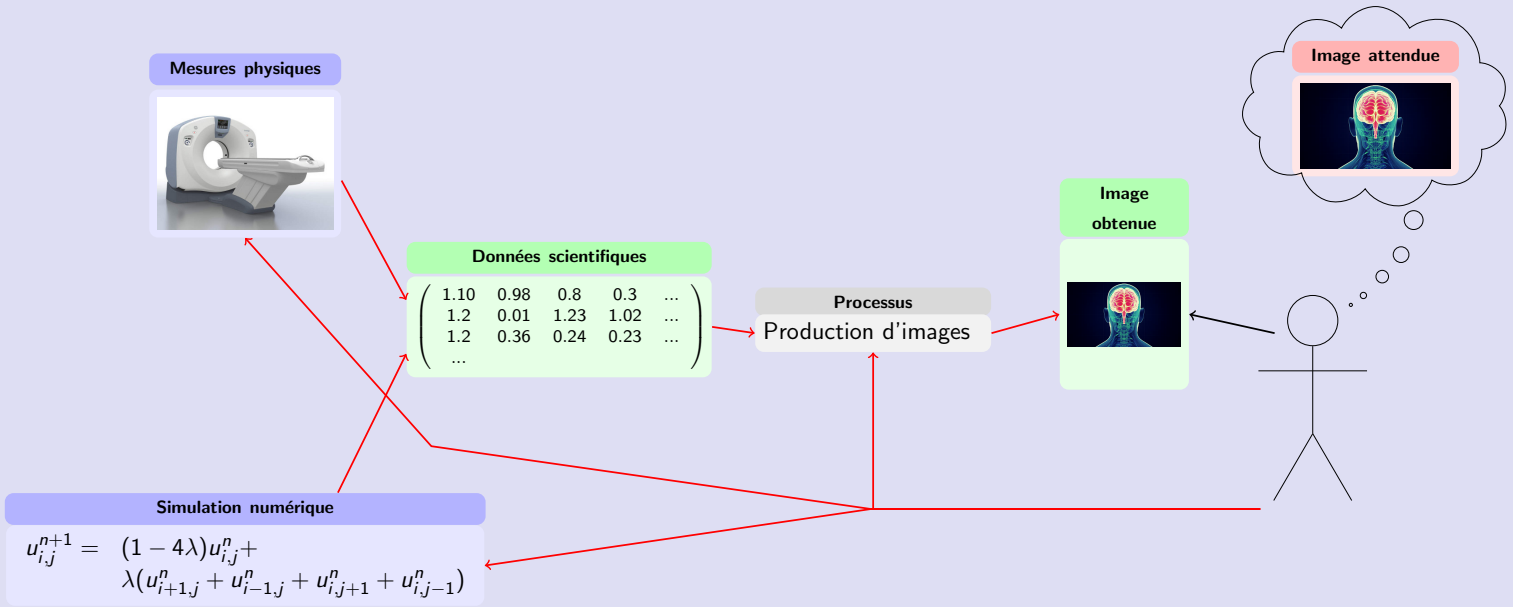
Boucle d'interaction visuelle



Si ce n'est pas le cas, il modifie

- soit les paramètres du processus de visualisation,
- soit ceux de la simulation numérique,
- soit du processus d'acquisition.

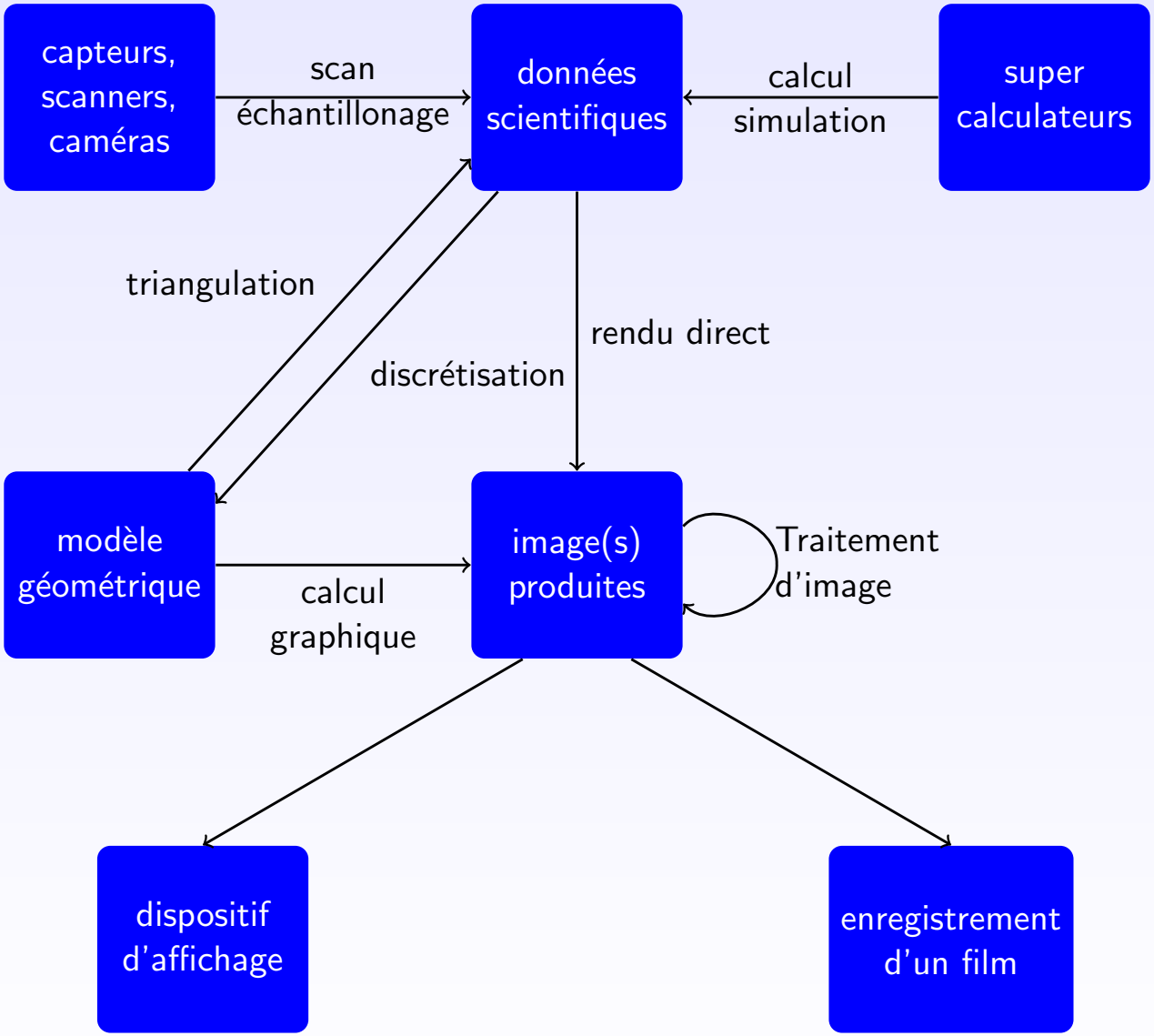
Boucle d'interaction visuelle



Pour être intéressante du point de vue scientifique il faut que

- les représentations graphiques soient représentatives des phénomènes ou objets observés,
- la boucle soit la plus rapide possible

Processus de production des images



Les données

Les données issues de scanners ou de capteurs

- Les informations sont souvent de la forme (x, y, z, val) où (x, y, z) est la position en 3D de la mesure et val est la quantité mesurée (densité, concentration ...).
- Suivant le type d'appareil les coordonnées
  - régulières c-à-d les points de mesures sont régulièrement répartis dans l'espace
  - indépendantes c-à-d les points de mesures sont répartis de manière "aléatoire" dans l'espace

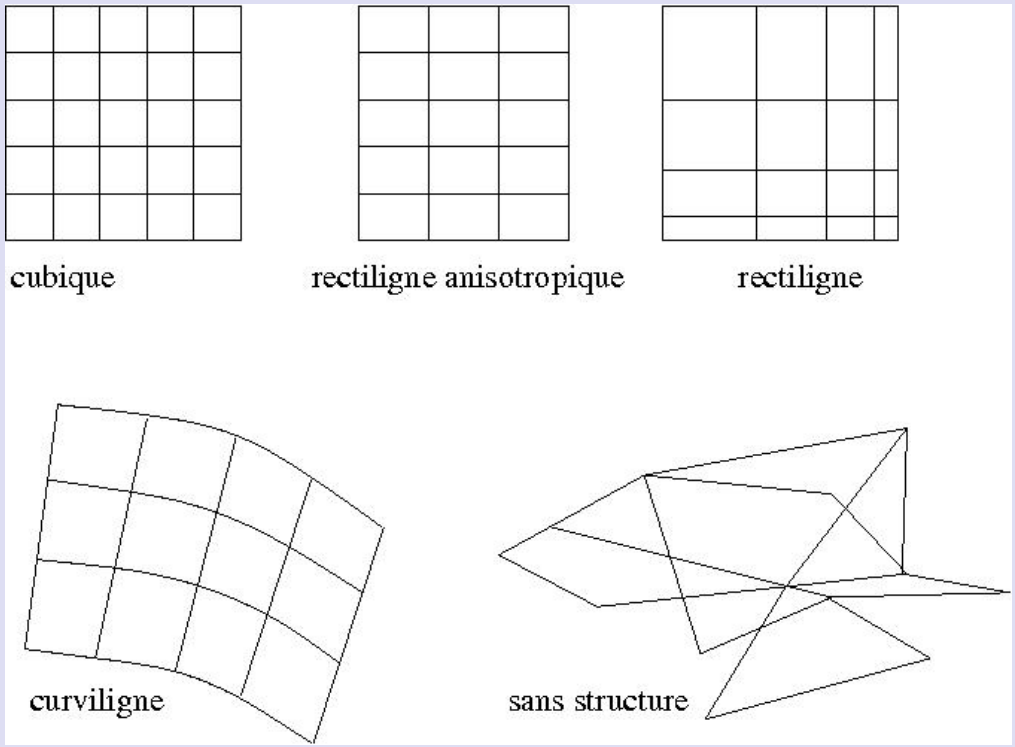
Les données issues de simulations numériques

- Les simulations numériques reposant sur la résolution d'équations aux dérivées partielles
  - A chaque pas de temps on obtient pour chaque élément (x, y, z) les valeurs des différentes quantités simulées
  - On obtient des valeurs (x, y, z, val) pour un domaine discrétisé de manière plus ou moins régulière
- Les simulations numériques de type N-corps
  - Ici on calcul à chaque pas de temps, un certain nombre de paramètres (vitesse, position, ...) sur des corps qui interagissent entre eux.
  - Données très irrégulières $\Rightarrow$ rendu très différent des cas précédents

Types de grilles

En général grilles régulières

- Le voisinage est implicite
- Pour les simulations numériques, les grilles ne sont pas nécessairement régulières mais souvent on s'y ramène.
- Pour les données irrégulières issues de mesures, on se ramène souvent à du régulier par interpolation sur une grille régulière



Les données

Données discrètes

Elles sont données sous la forme d'un nuage de points

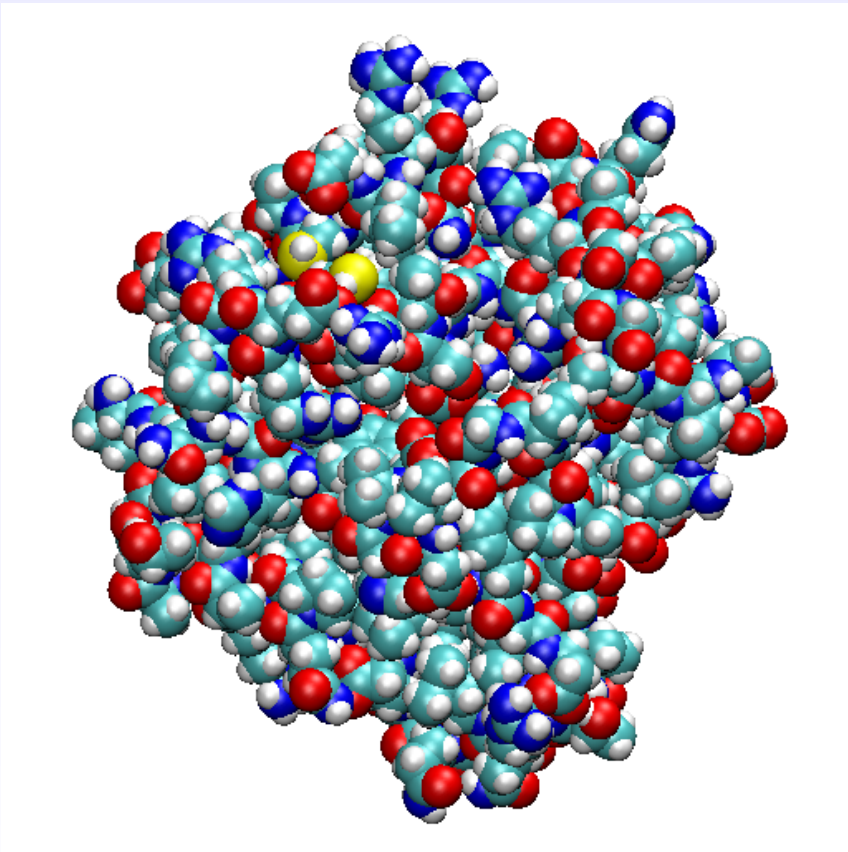
- La simulation numérique fournie une matrice 3D de résultats à visualiser.
- Imagerie médicale directement des données volumiques par intensité de gris
- Systèmes bio moléculaires des positions d'atomes
- Modèle numérique de terrains des relevés sur un maillage avec un pas donné
- ....

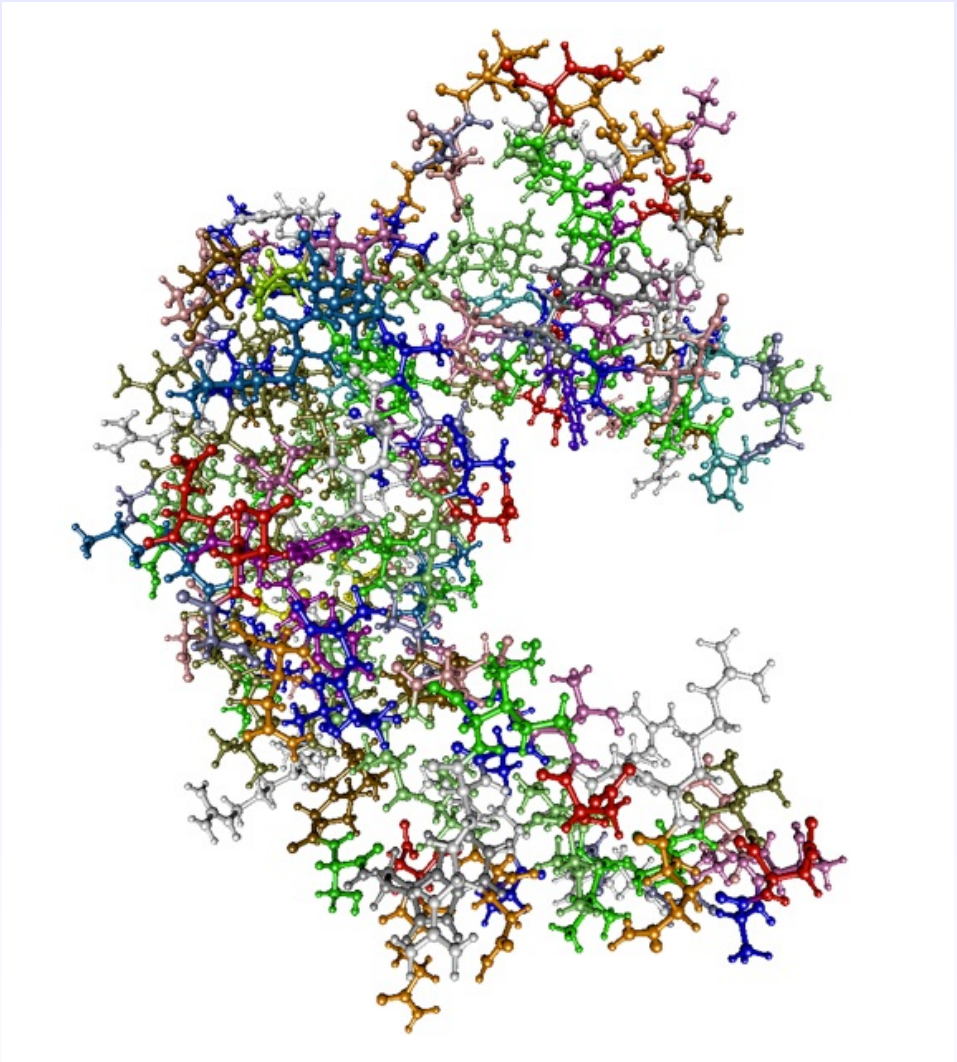
Enjeux

Visualiser ces données pour faciliter leur compréhension

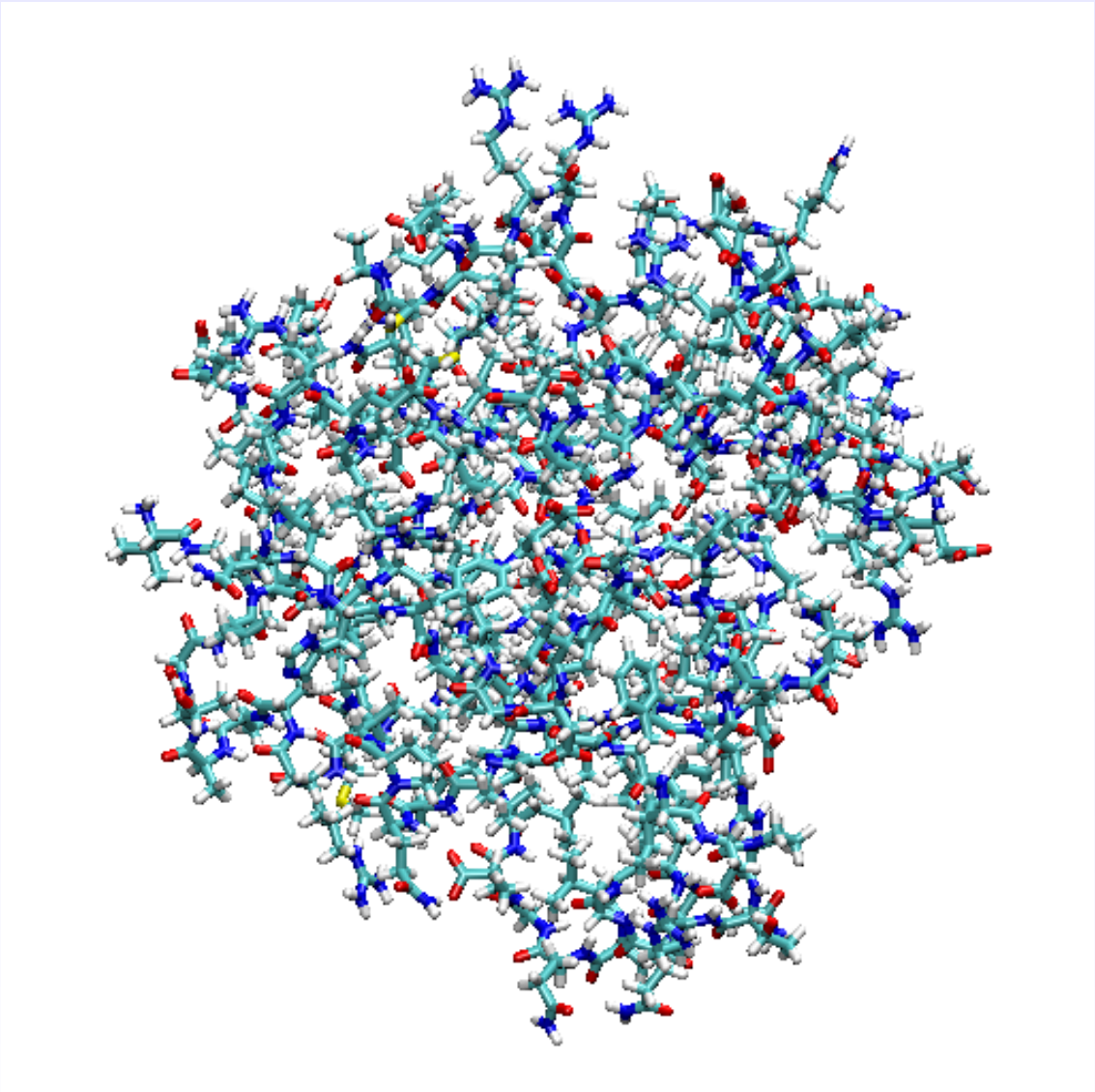
- Trouver une représentation graphique des données
- Mettre en exergue les aspects intéressants des données

Exemples : Systèmes bio moléculaires

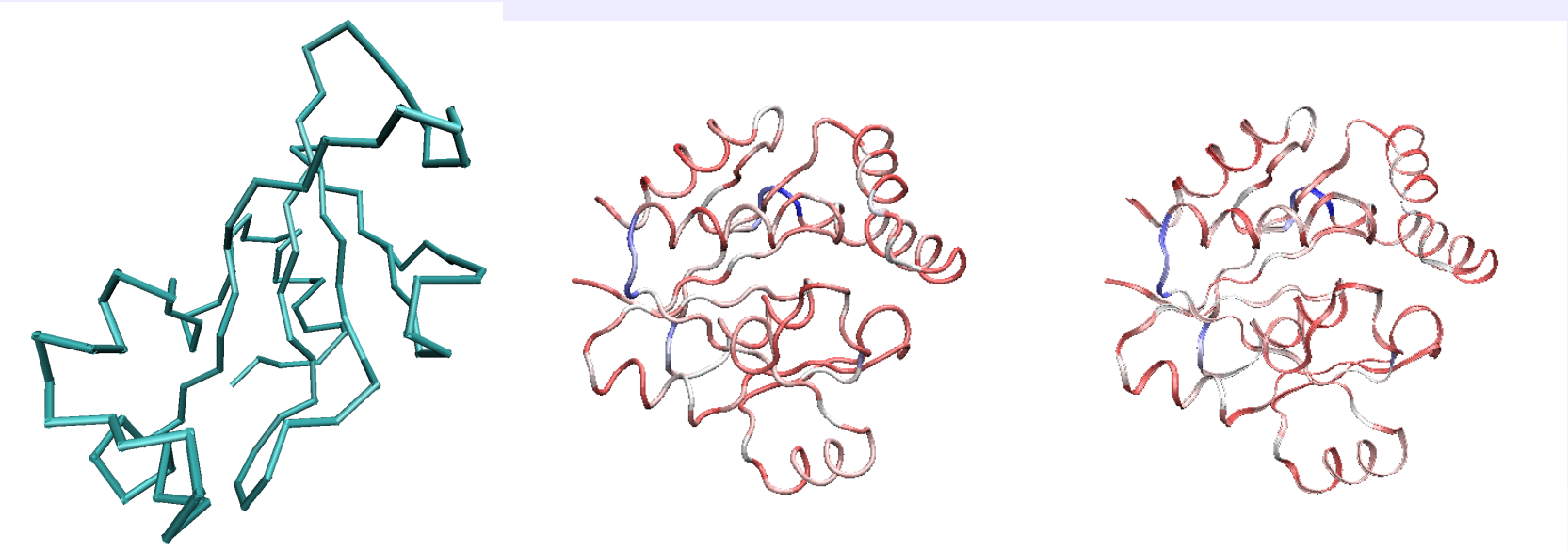




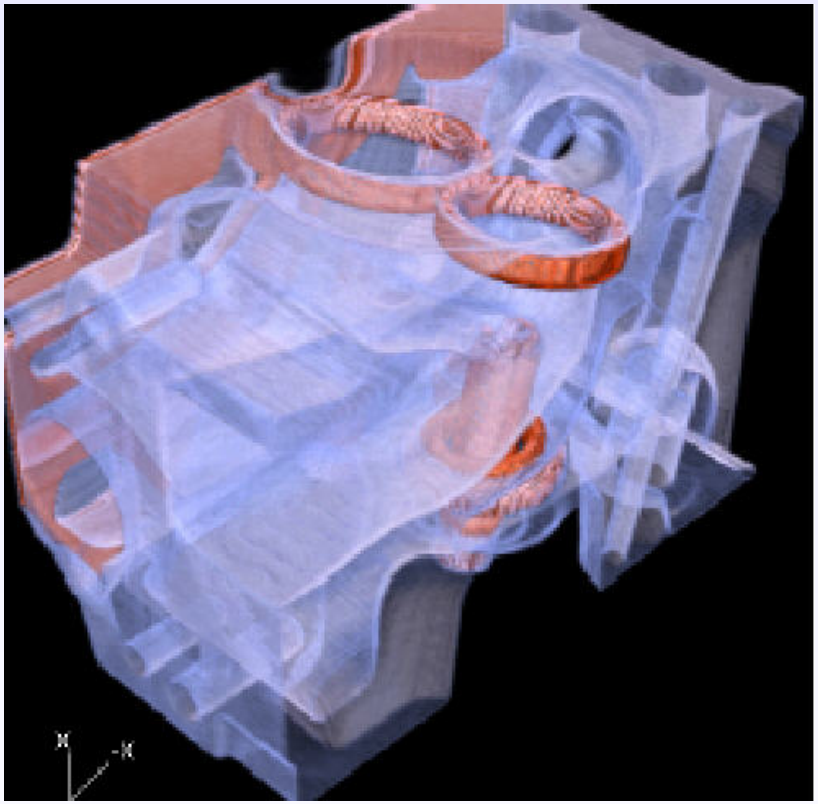
Exemples : Systèmes bio moléculaires



Exemples : Systèmes bio moléculaires

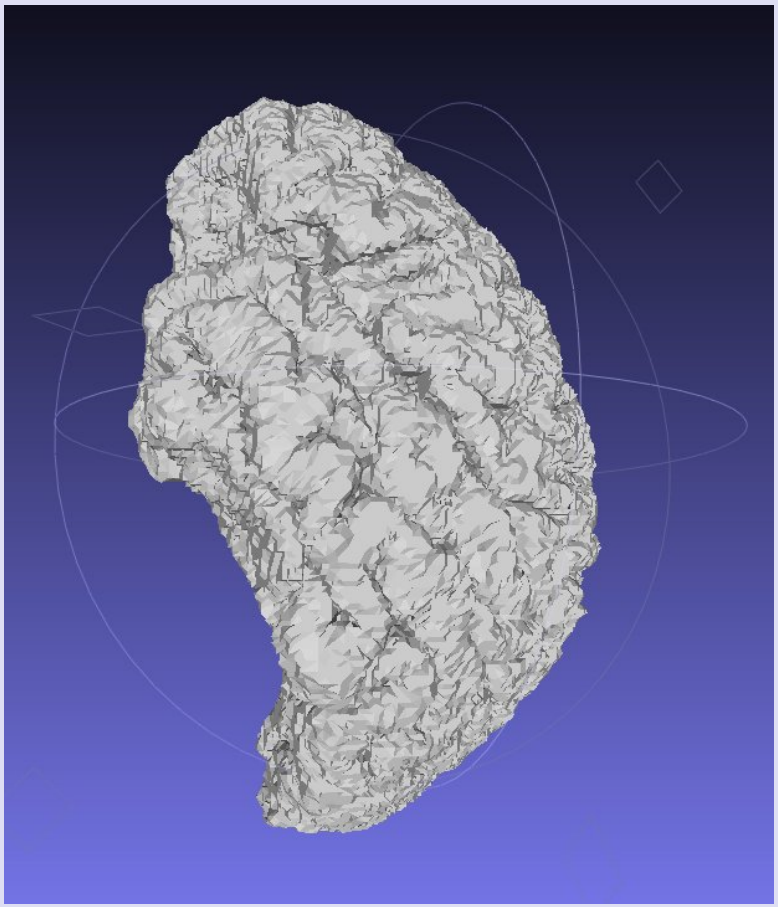


Exemple : Rendus de scanner



Exemple : Imagerie Médicale (http://brainder.org)

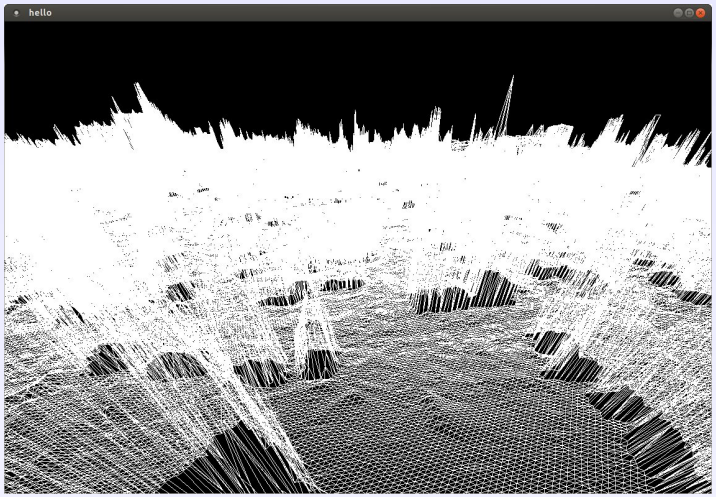
Hémisphère gauche d'un cerveau

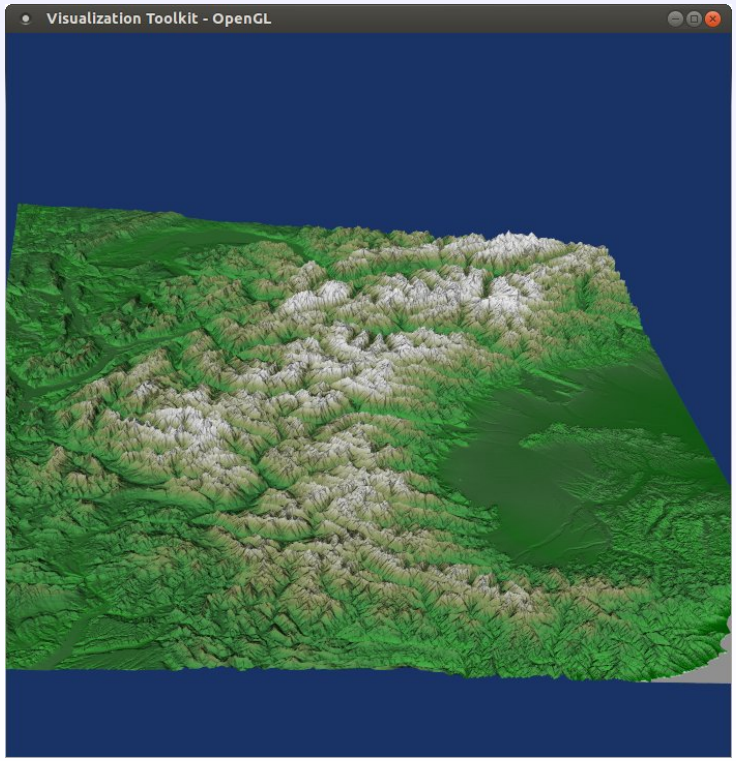


Les données

- un ensemble de voxel de $0.8 \times 0.8 \times 0.8$ millimetres
- mesh généré : 144490 points
- triangulation : 180722 facets

Exemple : MNT





Les données (La Mouillière)

- Une dalle de 1025×1025
- $\sim 1 km^2$

Les données (Les alpes)

- Une dalle de 1024×1024
- $\sim 10000 km^2$

Outils pour la visualisation scientifique

Quelques outils

- VTK et Paraview (interface graphique au dessus de VTK) Los Alamos National Laboratory et Kitware
- Visit Lawrence Livermore National Laboratory
- Ensight CEISoftware
- etc...

Visualisation scientifique en OpenGL

- Retour sur le pipeline graphique
- Techniques à base de shaders pour la visualisation scientifique

Plan

Algorithme Général

Pour chaque image on fait les opérations suivantes

1. Effacer le buffer d'affichage (Framebuffer)
2. Positionner les paramètres de la scène (point de vue, lumière, etc...)
3. Pour chaque objet de la scène
   1. Charger la géométrie et les paramètres de couleurs
   2. Rastériser l'objet dans le Framebuffer (dans le pipeline)
4. Afficher l'image calculée à l'écran

Rendu graphique en OpenGL: exemple

```
  void Display(void){
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW); glLoadIdentity();
    glPushMatrix();
    glColor3f(1,1,0); glTranslatef(2,3,0);
    glutWireTeapot(0.5); // lancement du pipeline
    glPopMatrix();

    glColor3f(1,0,0);
    glBegin(GL_QUADS);
     glVertex3f(-0.55,-2,2.); glVertex3f( 0.55,-2,2.);
     glVertex3f( 0.55,-0.5,2.); glVertex3f(-0.55,-0.5,2.);
    glEnd(); // lancement du pipeline
   glutSwapBuffers();
}
```

Le Pipeline Graphique

Définition

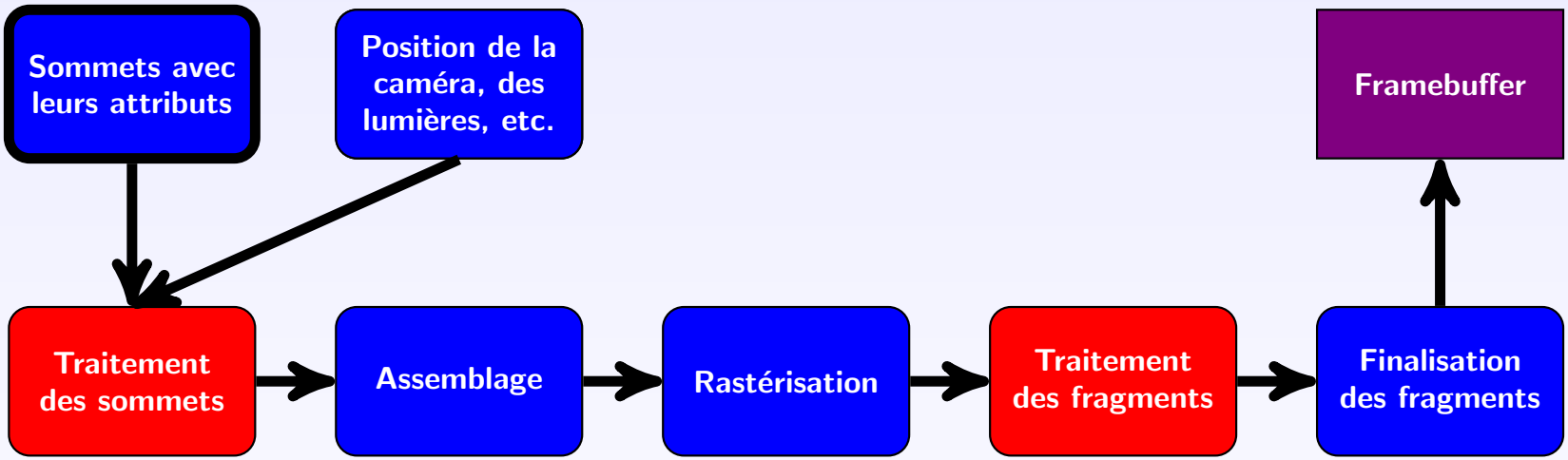
C'est l'ensemble des opérations qui à partir d'un objet décrit en 3D produit une image 2D (un tableau de pixels).

- Entrées:
  - Point de vue et cône de vision
  - La zone d'affichage à l'écran (ViewPort)
  - Les sources de lumières
  - Les textures
  - La géométrie de l'objet à dessiner
  - Les propriétés des sommets de cette géométrie (matériaux, normales, coordonnées de texture, etc...)
  - Etats de la machine OpenGL
- Sortie:
  - Une image 2D (c'est à dire un tableau de pixels de la taille de la zone écran à afficher)

Le Pipeline Graphique

Définition

C'est l'ensemble des opérations qui à partir d'un objet décrit en 3D produit une image 2D (un tableau de pixels).

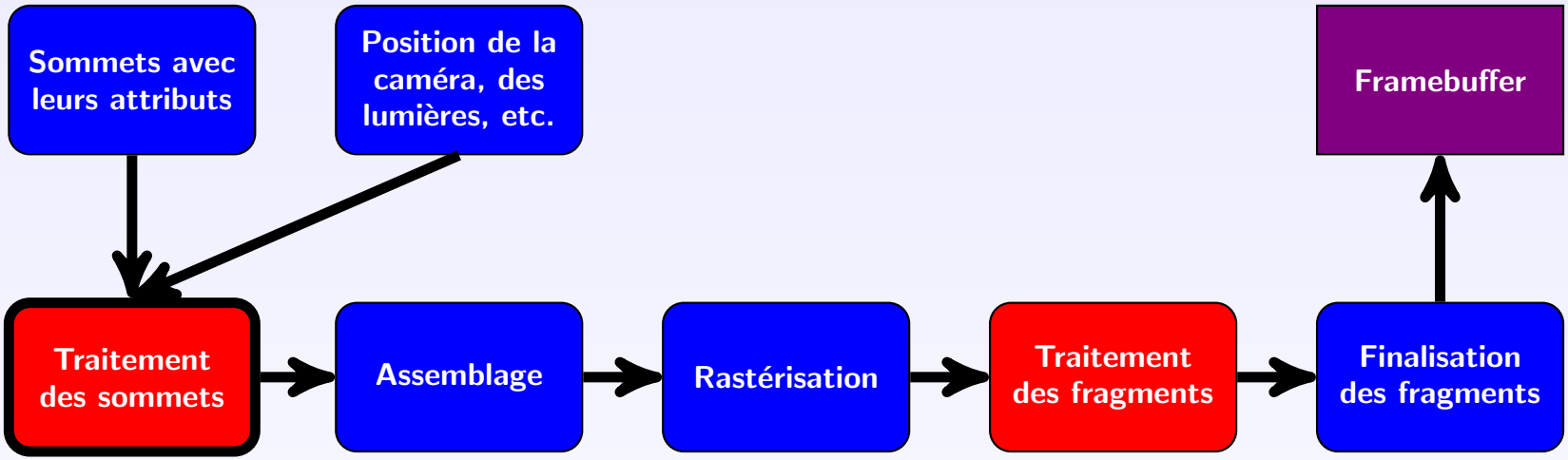


- Chargement des sommets avec leurs attributs
- Avec les primitives graphiques OpenGL
- Utilisation de tableaux ou de VBO

Le Pipeline Graphique

Définition

C'est l'ensemble des opérations qui à partir d'un objet décrit en 3D produit une image 2D (un tableau de pixels).

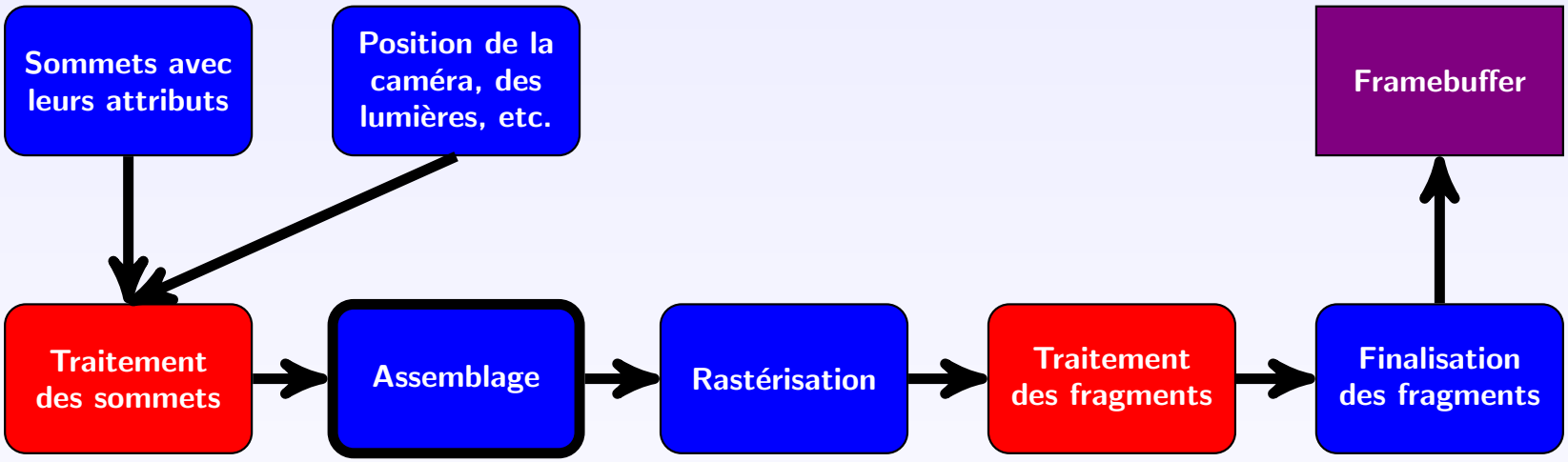


- Calcul de la position des sommets dans l'espace écran
- Cette étape peut être programmée (vertex Shader) pour modifier
  - les attributs des sommets
  - les plans de clipping
  - la taille du point

Le Pipeline Graphique

Définition

C'est l'ensemble des opérations qui à partir d'un objet décrit en 3D produit une image 2D (un tableau de pixels).

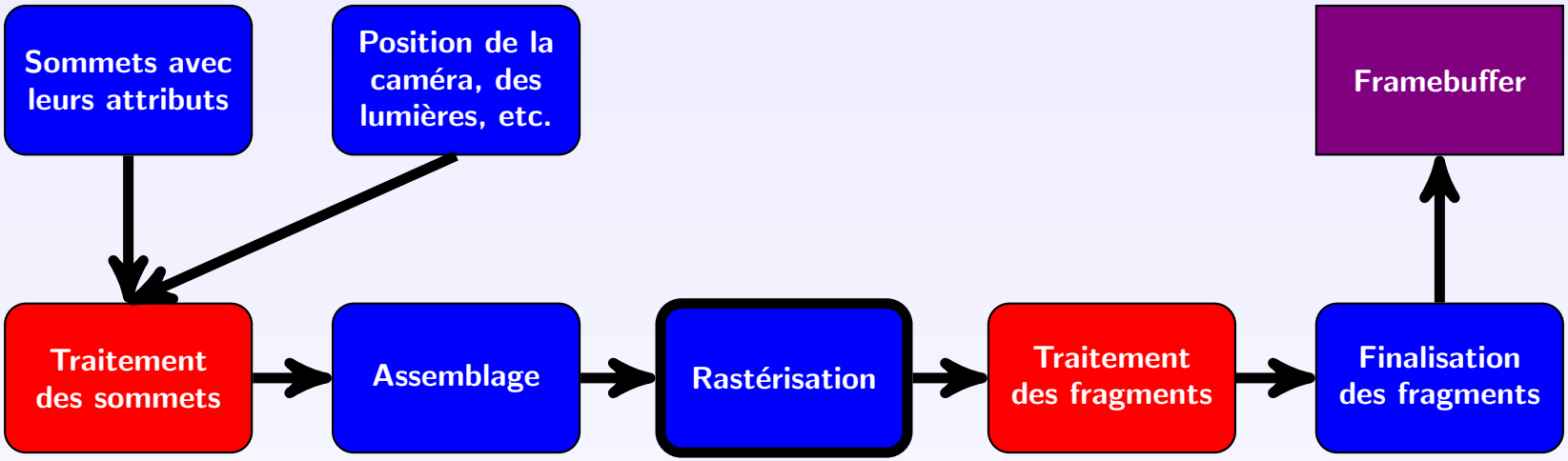


- Regroupement des sommets en polygones (faces)
- Cette étape n'est pas programmable

Le Pipeline Graphique

Définition

C'est l'ensemble des opérations qui à partir d'un objet décrit en 3D produit une image 2D (un tableau de pixels).

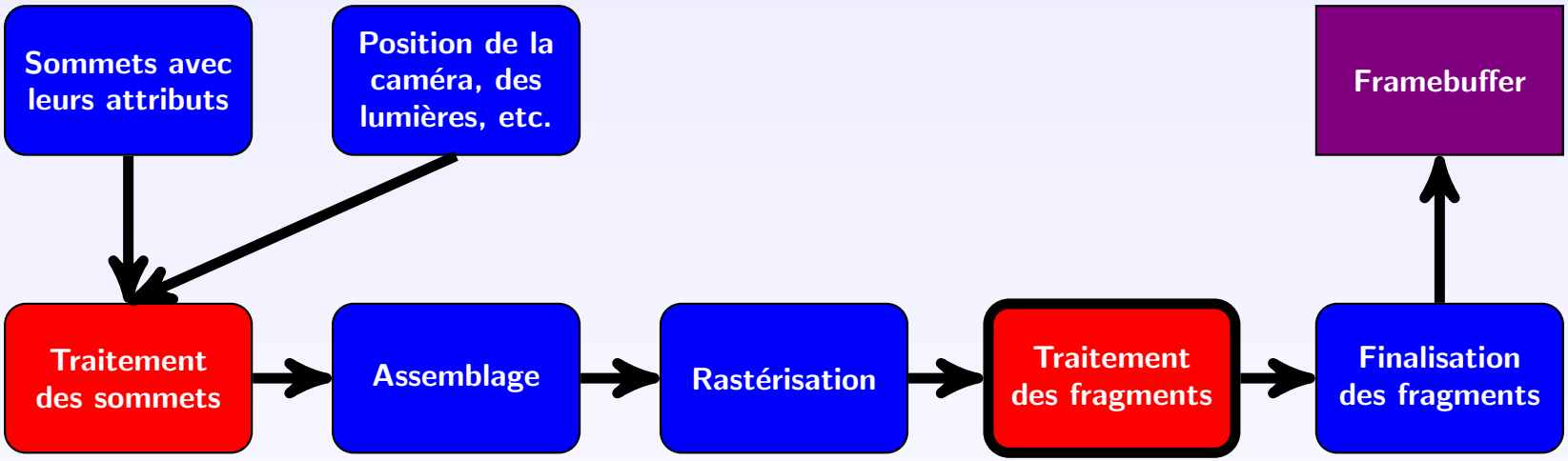


- Projection de chaque facette sur les pixels de l'espace écran
- Interpolation des couleurs, coordonnées de texture, profondeurs etc...
- Un pixel est appelé fragment
- Cette étape n'est pas programmable

Le Pipeline Graphique

Définition

C'est l'ensemble des opérations qui à partir d'un objet décrit en 3D produit une image 2D (un tableau de pixels).

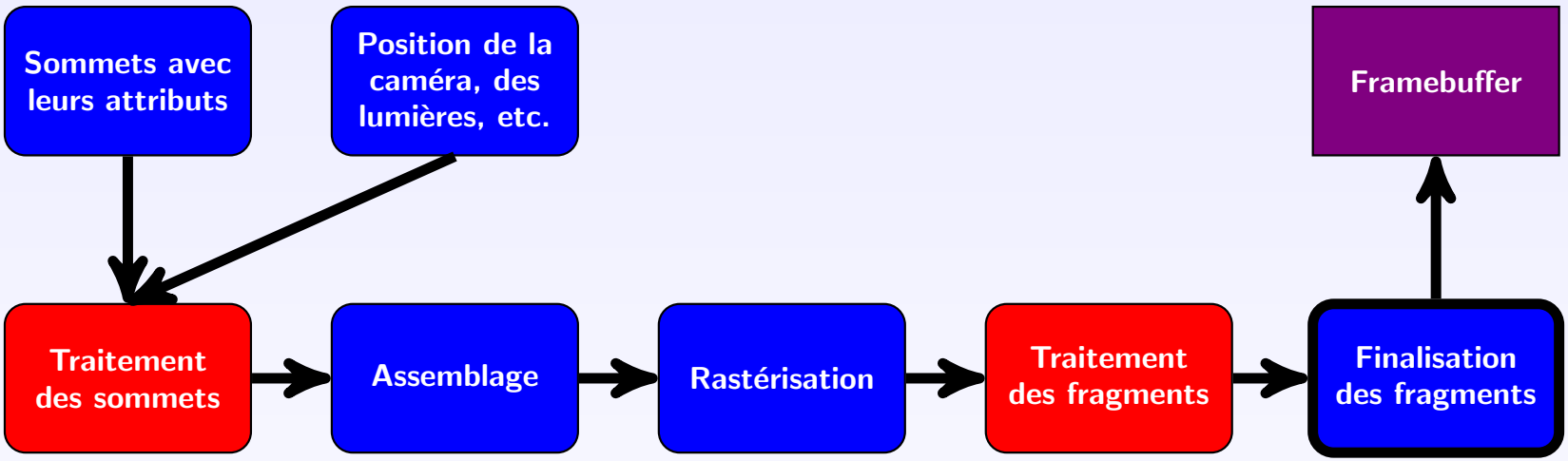


- Calcul de la couleur du fragment
- Calcul de sa profondeur
- Cette étape est programmable

Le Pipeline Graphique

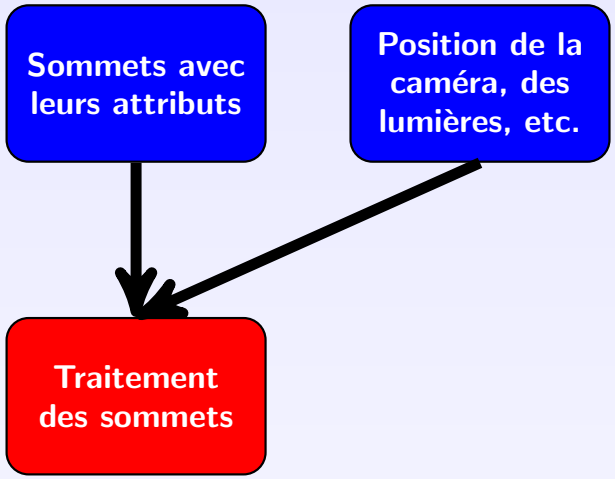
Définition

C'est l'ensemble des opérations qui à partir d'un objet décrit en 3D produit une image 2D (un tableau de pixels).



- Combinaison du fragment dans le framebuffer
  - Test de profondeur
  - Transparence
- Cette étape n'est pas programmable

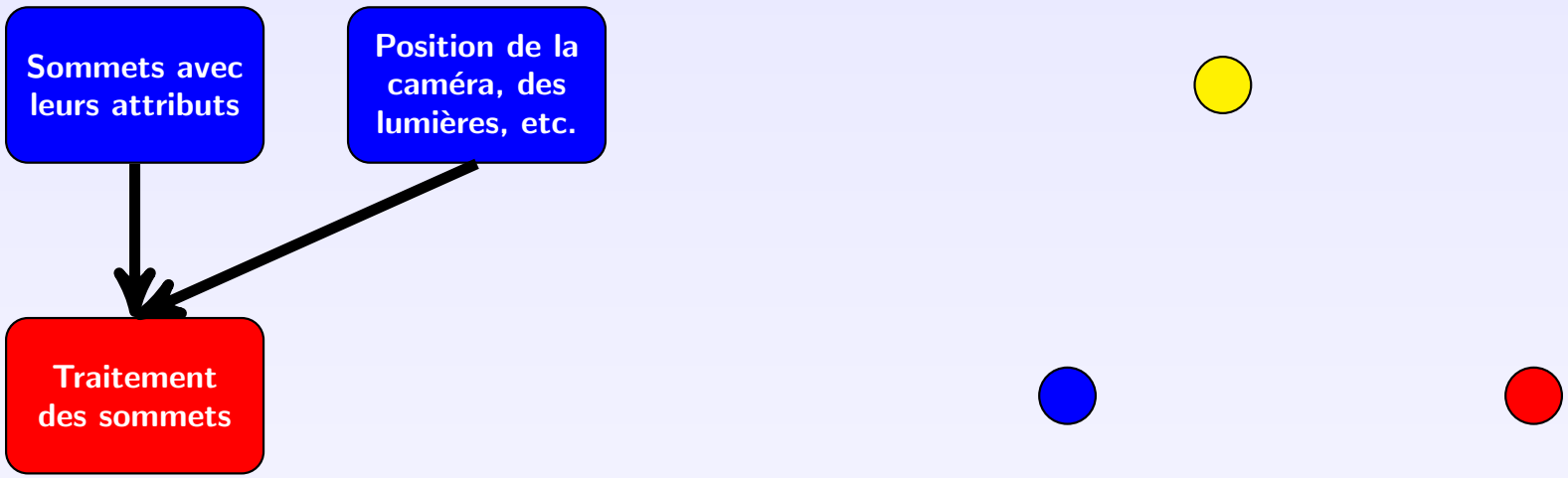
Traitement des sommets



Quelques détails

- Les paramètres généraux sont mis à jour dans la carte graphique
- Les sommets sont chargés dans la carte graphique
- Les attributs sont affectés à chacun des sommets

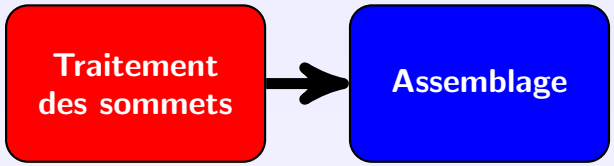
Traitement des sommets

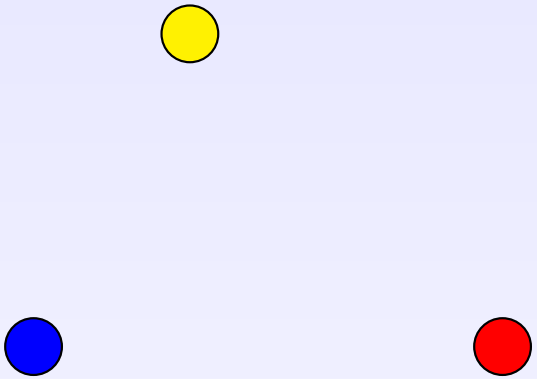


Quelques détails

- Les paramètres généraux sont mis à jour dans la carte graphique
- Les sommets sont chargés dans la carte graphique
- Les attributs sont affectés à chacun des sommets

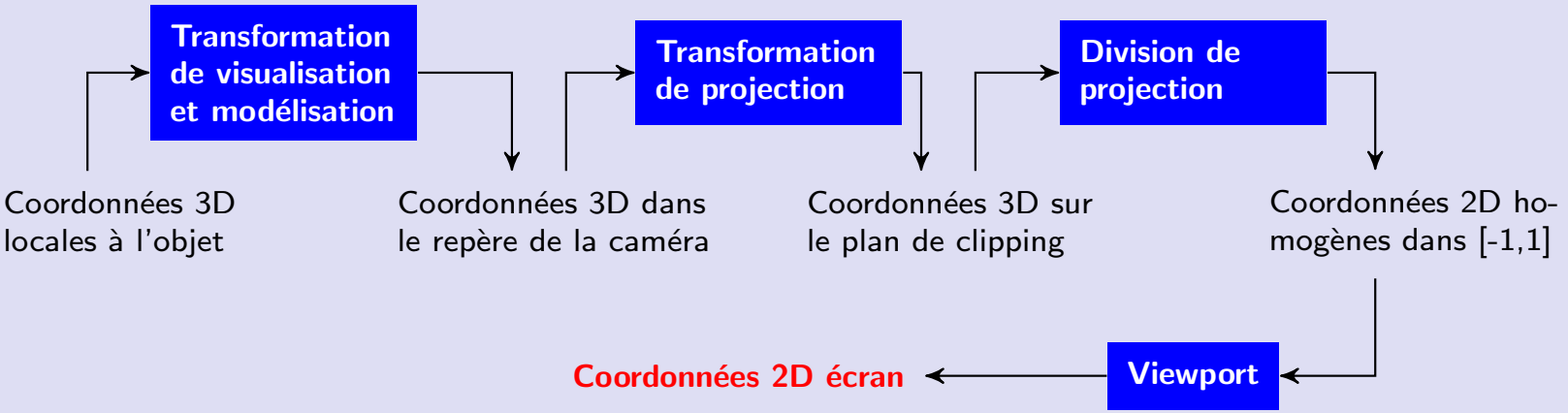
Assemblage des sommets



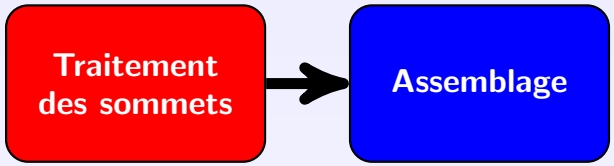


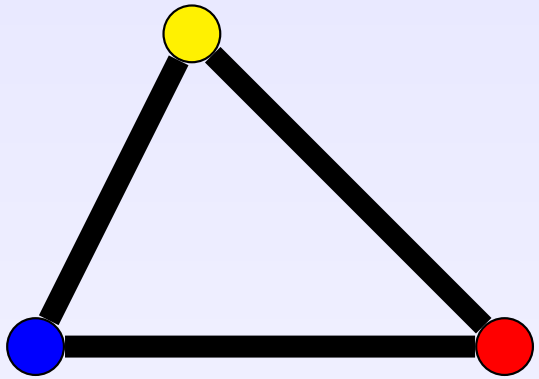
Quelques détails

Passage des coordonnées locales de l'objet en coordonnées écran



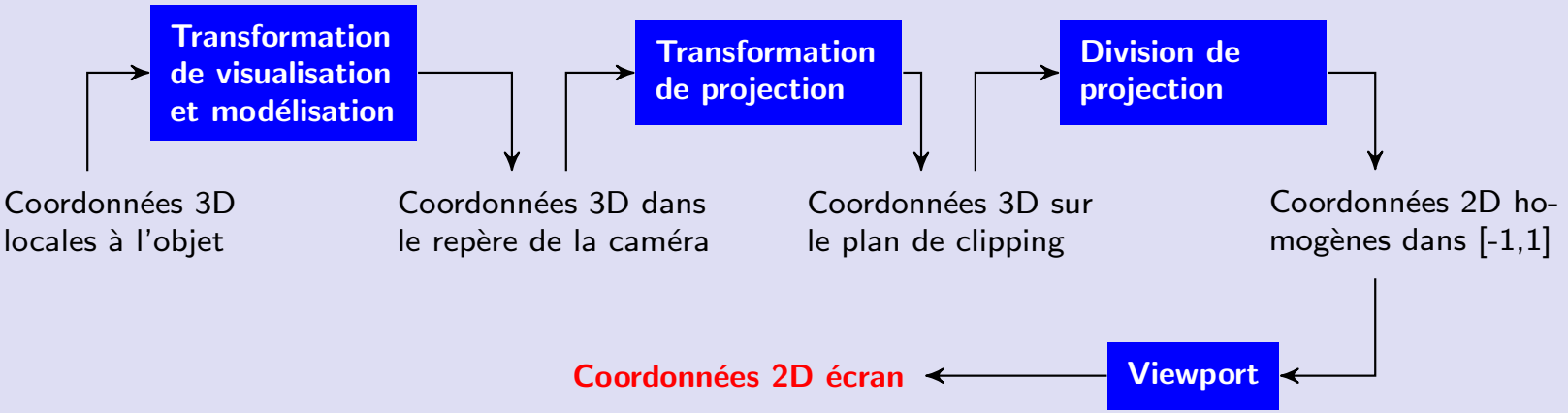
Assemblage des sommets





Quelques détails

Passage des coordonnées locales de l'objet en coordonnées écran

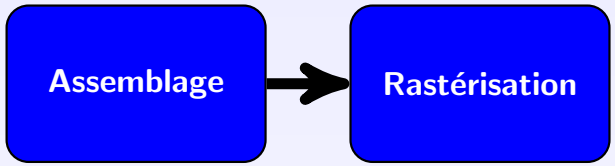


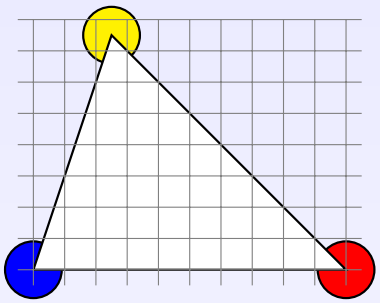
Rastérisation

Quelques détails

- Chaque polygone est pixelisé dans la résolution de l'écran
- Un pixel de la rastérisation est appelé fragment
- Les attributs (normale, couleur, profondeur,etc...) des sommets sont interpolés pour les fragments

Rastérisation

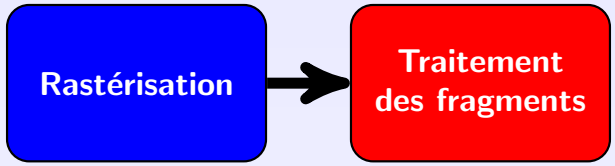




Quelques détails

- Chaque polygone est pixelisé dans la résolution de l'écran
- Un pixel de la rastérisation est appelé fragment
- Les attributs (normale, couleur, profondeur,etc...) des sommets sont interpolés pour les fragments

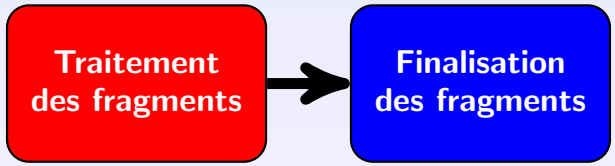
Traitement des fragments



Quelques détails

- Modification de la couleur du pixel
  - application des textures
  - application des éclairages
  - effet brouillard
  - etc...
- En sortie on a pour chaque fragment
  - sa couleur en RGBA
  - sa profondeur

Finalisation des fragments



Quelques détails

- Les fragments vont être combinés avec les pixels du framebuffer pour apporter leur contribution finale à l'image
  - Calcul de la profondeur relative
  - Application des transparences (blending)
  - Détermination des cas limites

Plan

Introduction

Objectifs

- Optimiser le nombre de sommets envoyés au GPU
- Utiliser au mieux le parallélisme du GPU

$\Longrightarrow$ Améliorer les performances du rendu

Plan

- Bref rappel sur le dessin *classique*
- Bref rappel sur les tableaux de sommets
- Les VBO – mise en œuvre

Dessin OpenGL

Rappels

- Appel à une des primitives de dessin (`GL_POINTS`, `GL_TRIANGLES`, `GL_QUAD`,, etc...
- Pour chaque sommet
  - Définition des propriétés (couleurs, normales, coordonnées de texture)
  - Positionnement dans l'espace
- `gl_End();` lance le pipeline

Dessin OpenGL: exemple du cube

```
glBegin( GL_TRIANGLES );
 glColor3f( 1, 0, 0 ); glVertex3f( -1, 1, -1 );  glColor3f( 1, 0, 1 ); glVertex3f( -1, -1, -1 );
 glColor3f( 1, 1, 1 ); glVertex3f( -1, 1, 1 );   glColor3f( 1, 1, 1 ); glVertex3f( -1, 1, 1 );
 glColor3f( 1, 0, 1 ); glVertex3f( -1, -1, -1 ); glColor3f( 0, 0, 1 ); glVertex3f( -1, -1, 1 );
 glColor3f( 0, 1, 0 ); glVertex3f( 1, 1, 1 );    glColor3f( 0, 1, 1 ); glVertex3f( 1, -1, 1 );
 glColor3f( 1, 1, 0 ); glVertex3f( 1, 1, -1 );   glColor3f( 1, 1, 0 ); glVertex3f( 1, 1, -1 );
 glColor3f( 0, 1, 1 ); glVertex3f( 1, -1, 1 );   glColor3f( 1, 1, 1 ); glVertex3f( 1, -1, -1 );
 glColor3f( 0, 0, 1 ); glVertex3f( -1, -1, 1 );  glColor3f( 1, 0, 1 ); glVertex3f( -1, -1, -1 );
 glColor3f( 0, 1, 1 ); glVertex3f( 1, -1, 1 );   glColor3f( 0, 1, 1 ); glVertex3f( 1, -1, 1 );
 glColor3f( 1, 0, 1 ); glVertex3f( -1, -1, -1 ); glColor3f( 1, 1, 1 ); glVertex3f( 1, -1, -1 );
 glColor3f( 1, 0, 0 ); glVertex3f( -1, 1, -1 );  glColor3f( 1, 1, 1 ); glVertex3f( -1, 1, 1 );
 glColor3f( 1, 1, 0 ); glVertex3f( 1, 1, -1 );   glColor3f( 1, 1, 0 ); glVertex3f( 1, 1, -1 );
 glColor3f( 1, 1, 1 ); glVertex3f( -1, 1, 1 );   glColor3f( 0, 1, 0 ); glVertex3f( 1, 1, 1 );
 glColor3f( 1, 1, 0 ); glVertex3f( 1, 1, -1 );   glColor3f( 1, 1, 1 ); glVertex3f( 1, -1, -1 );
 glColor3f( 1, 0, 0 ); glVertex3f( -1, 1, -1 );  glColor3f( 1, 0, 0 ); glVertex3f( -1, 1, -1 );
 glColor3f( 1, 1, 1 ); glVertex3f( 1, -1, -1 );  glColor3f( 1, 0, 1 ); glVertex3f( -1, -1, -1 );
 glColor3f( 1, 1, 1 ); glVertex3f( -1, 1, 1 );   glColor3f( 0, 0, 1 ); glVertex3f( -1, -1, 1 );
 glColor3f( 0, 1, 0 ); glVertex3f( 1, 1, 1 );    glColor3f( 0, 1, 0 ); glVertex3f( 1, 1, 1 );
 glColor3f( 0, 0, 1 ); glVertex3f( -1, -1, 1 );  glColor3f( 0, 1, 1 ); glVertex3f( 1, -1, 1 );
glEnd();
```

- Beaucoup de code
- Répétition des sommets
- Rapidement inefficace

Les tableaux de sommets (Vertex Array)

Idée

- Stocker les sommets dans un tableau (Vertex Array) en mémoire CPU
- Stocker les propriétés des sommets dans des tableaux adéquats (Color Array, Normal Array,etc...) en mémoire CPU
- Stocker dans un tableau d'indices (Vertex Pointer) l'ordre d'utilisation des sommets
- Lancer le dessin des primitives via ces tableaux (glDrawElements)

VA: exemple du cube

```
GLfloat VertexArray[24] = {-1,1,-1, -1,1,-1, -1,1,1, -1,-1,1, 1,1,1, 1,-1,1, 1,1,-1, 1,-1,-1};

GLfloat ColorArray[24] =  {  1,0,0,   1,0,1,  1,1,1,   0,0,1, 0,1,0,  0,1,1,  1,1,0,   1,1,1};

GLuint IndiceArray[36] = {0,1,2,2,1,3,4,5,6,6,5,7,3,1,5,5,1,7,
                          0,2,6,6,2,4,6,7,0,0,7,1,2,3,4,4,3,5};

glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_COLOR_ARRAY );

glVertexPointer( 3, GL_FLOAT, 0, VertexArray );
glColorPointer( 3, GL_FLOAT, 0, ColorArray );

glDrawElements( GL_TRIANGLES, 36, GL_UNSIGNED_INT, IndiceArray );

glDisableClientState( GL_COLOR_ARRAY );
glDisableClientState( GL_VERTEX_ARRAY );
```

- Réduction du nombre d'informations envoyées vers le GPU
- Réduction du nombre de fonctions utilisées

$\Longrightarrow$ Amélioration des performances

Les Vertex Buffer Objects

Idée

- Charger les tableaux précédents en GPU
- Optimiser l'utilisation du GPU
- Limiter les transferts CPU/GPU

VBO: Les étapes

Génération des buffers

- Même principe que les textures
- Réservation de buffers
  `glGenBuffers(GLsizei nbBuf,Guint *idents)`
  permet d'obtenir `nbBuf` identifiants de buffers
- Utilisation d'un buffer
  `glBind(GLenum type, GLuint buffer)`
  où `type` est soit
  - `GL_ARRAY_BUFFER` un tableau d'information sur les sommets
  - `GL_ELEMENT_ARRAY_BUFFER` un tableau d'indices
- La fonction `glDeleteBuffers` permet de supprimer des buffers

VBO: Les étapes

Chargement des buffers en GPU

```
void glBufferData(GLenum type, GLsizeiptr taille,
const GLvoid *donnees, GLenum util)
```

- `type`: soit `GL_ARRAY_BUFFER` soit `GL_ELEMENT_ARRAY_BUFFER`
- `taille` est le nombre d'octets à charger
- `donnees` est un pointeur sur la zone memoire CPU contenant les informations
- `util` est le type d'utilisation du buffer
  - `DRAW` le plus courant pour dessiner (acces en écriture), il existe aussi `READ` (acces en lecture) et `COPY` (acces en lecture/écriture)
  - `STATIC` un buffer modifié très rarement
  - `STREAM` un buffer modifié de temps en temps
  - `DYNAMIC` un buffer modifié très souvent

  Ex: `GL_STREAM_DRAW`

VBO: Les étapes

Utilisation des buffers

```
void gXXXPointer(GLint size, GLenum type, GLsizei
pas, const GLvoid *pointeur)
```

- `size`: le nombre d'éléments par sommet
- `type`: type des données (`GL_FLOAT`, `GL_INT`,...)
- `pas`: nombre d'octets entre deux sommets
- `pointeur` un pointeur sur le premier sommet

Un même buffer peut contenir plus d'informations sur les sommets (coordonnées, couleur, etc...)

VBO: l'exemple du cube (initialisations)

```
GLfloat VertexArray[24] =
            {-1,1,-1, -1,1,-1, -1,1,1, -1,-1,1, 1,1,1, 1,-1,1, 1,1,-1, 1,-1,-1};

GLfloat ColorArray[24] =
            {1,0,0, 1,0,1, 1,1,1, 0,0,1, 0,1,0, 0,1,1, 1,1,0, 1,1,1};

GLuint IndiceArray[36] = {0,1,2,2,1,3,4,5,6,6,5,7,3,1,5,5,1,7,
                          0,2,6,6,2,4,6,7,0,0,7,1,2,3,4,4,3,5};
// Génération des buffers
// CubeBuffers doit être un tableau déclaré globalement
glGenBuffers( 3, CubeBuffers );

// Buffer des coordonnées de vertex
glBindBuffer(GL_ARRAY_BUFFER, CubeBuffers[0]);
glBufferData(GL_ARRAY_BUFFER, 24*sizeof(float), VertexArray, GL_STATIC_DRAW);
// Buffer des couleurs de vertex
glBindBuffer(GL_ARRAY_BUFFER, CubeBuffers[1]);
glBufferData(GL_ARRAY_BUFFER, 24*sizeof(float), ColorArray, GL_STATIC_DRAW);

// Buffer d'indices
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, CubeBuffers[2]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, 36*sizeof(int), IndiceArray,
     GL_STATIC_DRAW);
```

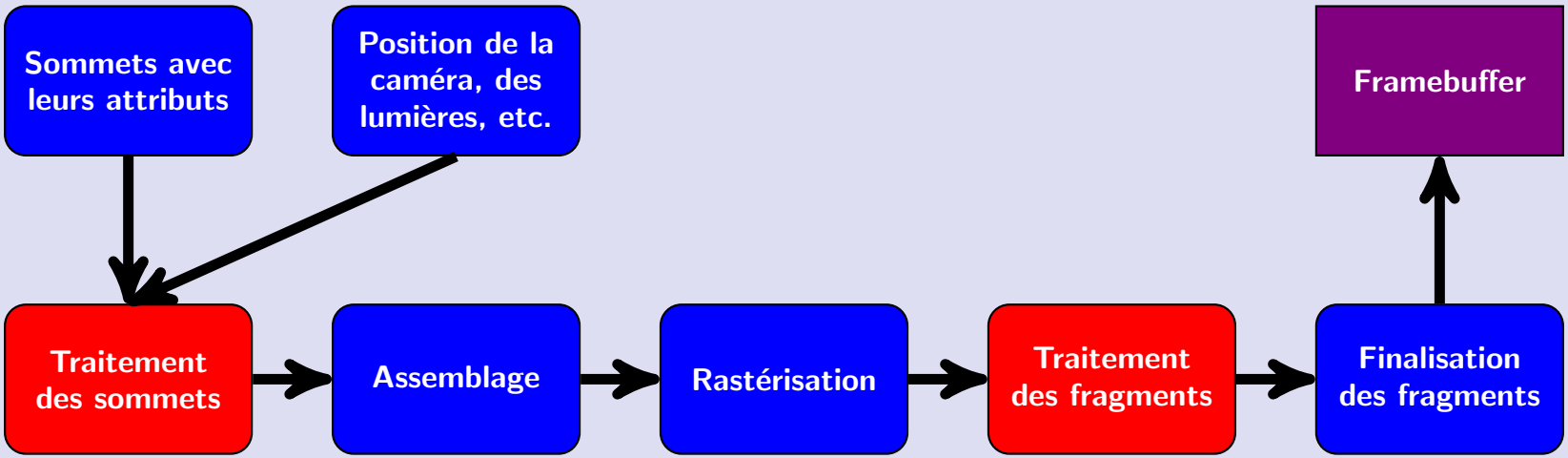
VBO: l'exemple du cube (dessin)

```
// Utilisation des données des buffers
glBindBuffer(GL_ARRAY_BUFFER, CubeBuffers[0]);
glVertexPointer( 3, GL_FLOAT, 3 * sizeof(float), 0 );
glBindBuffer(GL_ARRAY_BUFFER, CubeBuffers[1]);
glColorPointer( 3, GL_FLOAT, 3 * sizeof(float), 0 );

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, CubeBuffers[2]);

// Activation d'utilisation des tableaux
glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_COLOR_ARRAY );

// Rendu de notre géométrie
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);

glDisableClientState( GL_COLOR_ARRAY );
glDisableClientState( GL_VERTEX_ARRAY );
```

Plan

Principes

- Remplacer le traitement par défaut d'Open GL dans le pipeline graphique
- Langage de programmation de haut niveau (plus ou moins) indépendent de la carte graphique



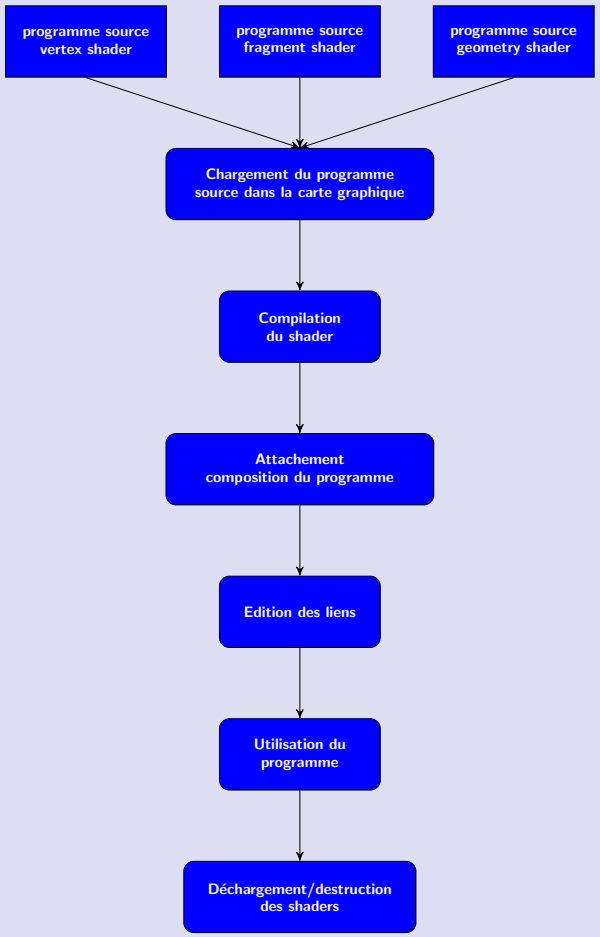
Les langages de shaders

Trois principaux langages

- GLSL OpenGL Shading Language: le langage développé par OpenGL
- CG: Langage développé par nVidia pour les cartes graphiques nVidia
- HLSL DirectX High-Level Shader Language: langage développé par Microsoft pour les environnements Microsoft

Shaders dans la pratique

Les étapes

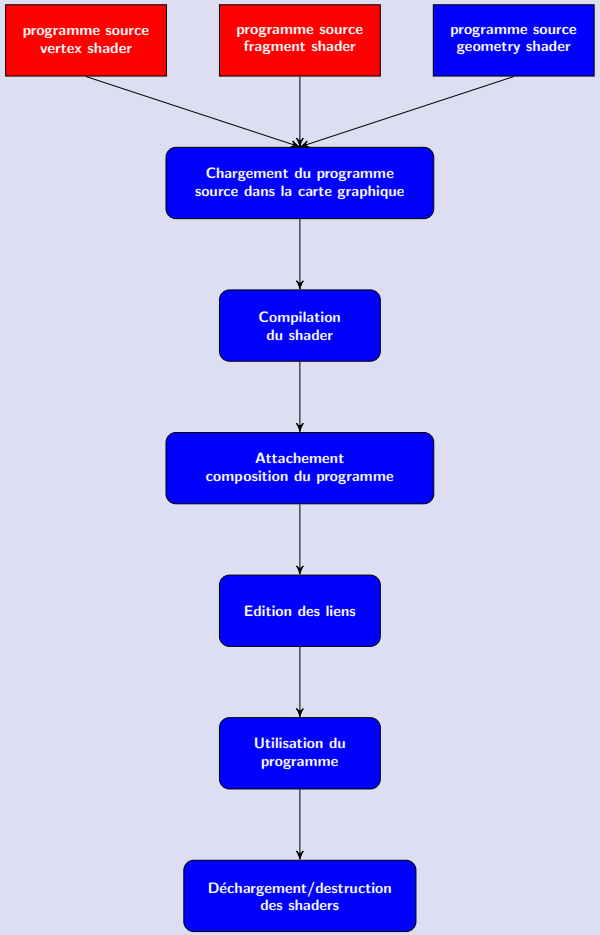


Vocabulaire

- Shader: C'est une des étapes du pipeline que l'on veut programmer
  - Vertex shader
  - Fragment shader
  - Geometry shader
  - ...
- Programme: C'est l'assemblement de plusieurs shaders de types différents qui définissent le nouveau pipeline graphique

Shaders dans la pratique

Les codes sources



Illustration

- Fichier texte: syntaxe proche du C++
  - Vertex shader

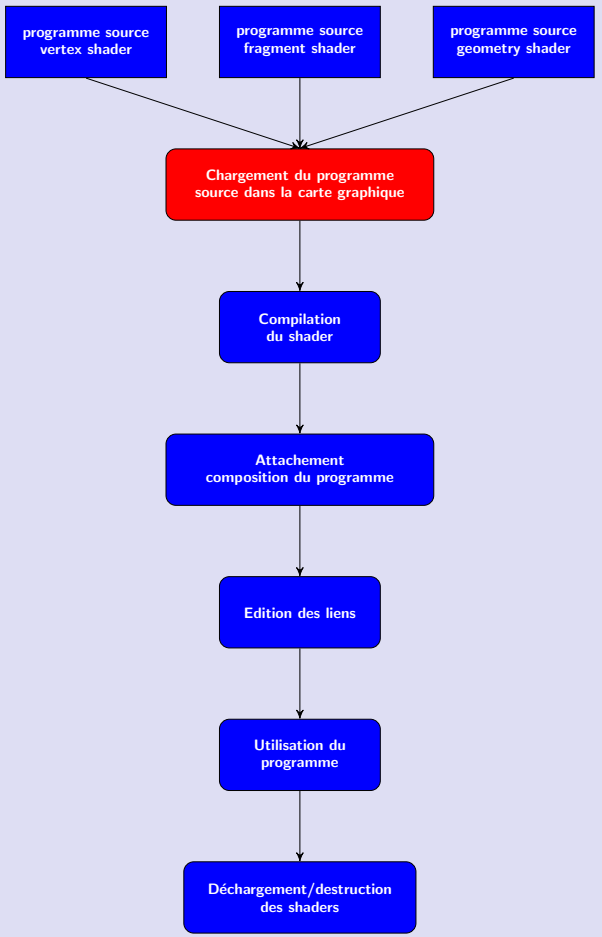
```
void main() {
  gl_Position =
   gl_ModelViewProjectionMatrix *
   gl_Vertex;
  gl_FrontColor = gl_Color; }
```

  - Fragment shader

```
void main() {
gl_FragColor = gl_Color;
}
```

- Ecrit indépendamment du programme C++ qui l'utilisera

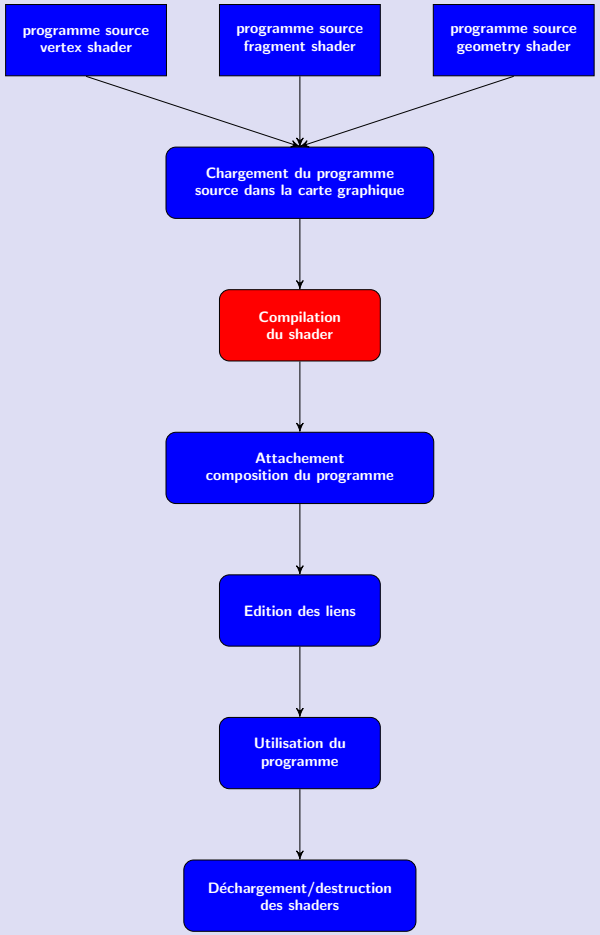
Chargement des sources



Illustration

- Il s'agit de charger le code source en carte graphique en vue de sa compilation
  - Création du shader
    `shId=glCreateShader(typeShader);`
    où `typeShader` est par exemple `GL_VERTEX_SHADER` ou `GL_FRAGMENT_SHADER`
    On récupère un identifiant de shader
  - Chargement du code source dans la carte graphique
    `glShaderSource(shId, nbSrc, src, NULL);`
    `src` est un tableau de chaines de caractères

La compilation



Illustration

- Transformation du shader source

  en assembleur de la carte graphique

  `glCompileShader (shadId);`
- Etat de la compilation

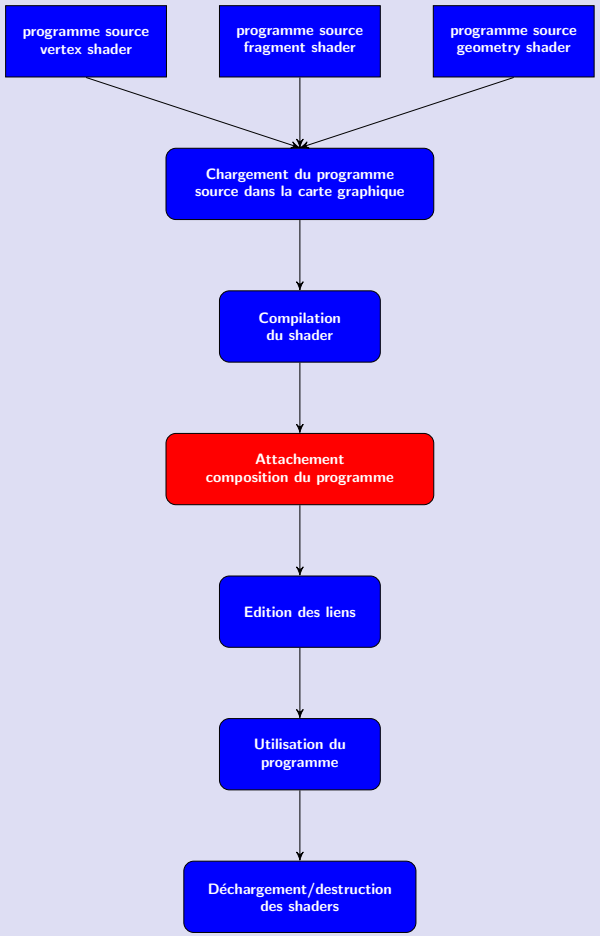
  `glGetShaderiv(shadId, GL_COMPILE_STATUS, &erreur);`

  `erreur` est un booléen indiquant si la compilation s'est bien passée
- Message d'erreur

  `glGetShaderInfoLog (shadId, taille, &lg, msg);`

  `msg` est un tableau de `taille` caractères, `lg` est la taille du message retourné

Composition du programme



Illustration

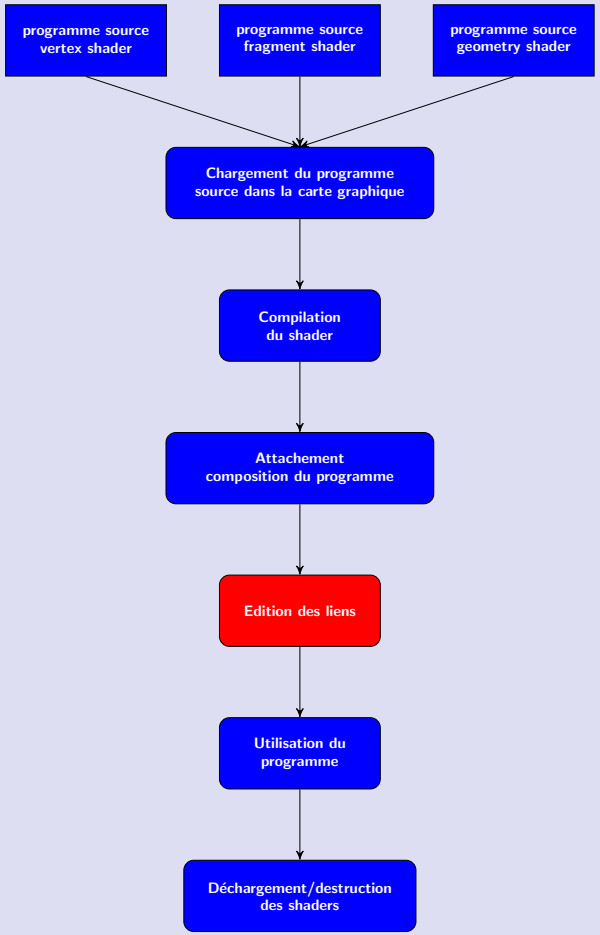
- Il s'agit d'assembler plusieurs shaders de types différents dans un programmes pour former un pipeline graphique.
- Création du programme

  ```
  shadProg=glCreateProgram();
  ```

  On récupère un identifiant de programme
- Attachement d'un shader à un programme.

  ```
  glAttachShader(shaderProg,shaderIdent);
  ```

Edition des liens



Illustration

- Cette étape consiste à mettre en place les mécanismes qui permettent aux shaders de se transmettre les données

```
glLinkProgram(shadPrg);
```

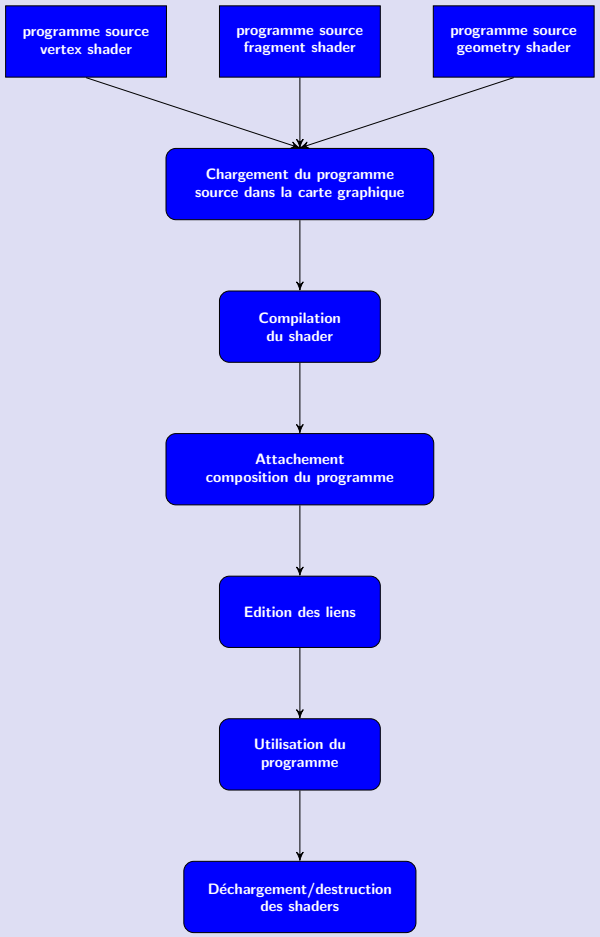
- Retour sur l'édition des liens

```
glGetShaderiv(shadPrg, GL_LINK_STATUS,
&erreur);
```

- Message d'erreur:

```
glGetProgramInfoLog (shadPrg, taille,
&lg, msg);
```

Utilisation du programme

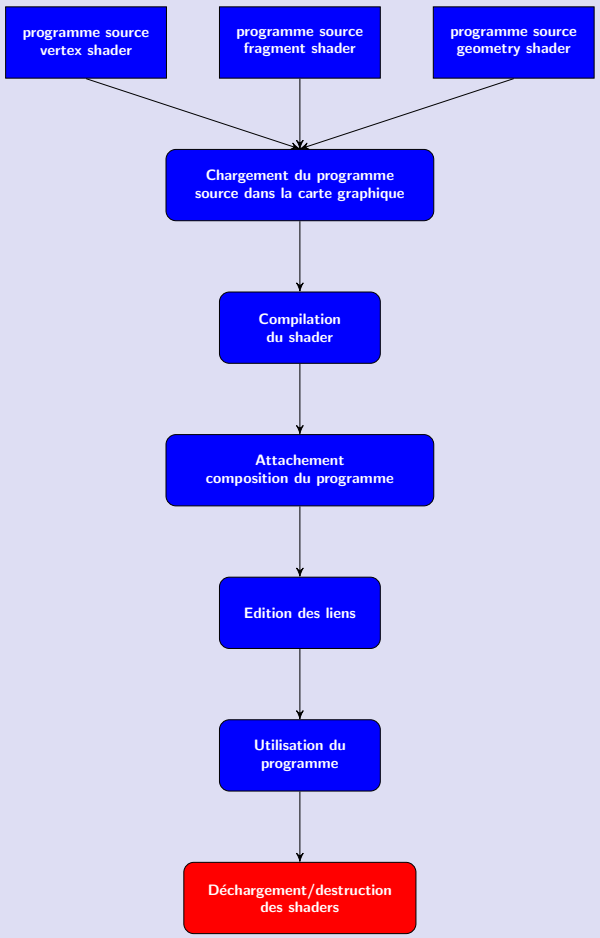


Illustration

- Juste avant le lancement du pipeline graphique il suffit d'appeler

```
glUseProgram (shaderProg);
```

Déchargement/destruction



Illustration

- Utilisé pour libérer les resources sur la carte graphique
  - `void glDetachShader(GLuint prog, GLuint shad);`
  - `void glDeleteShader(GLuint shadId);`
  - `void glDeleteProgram(GLuint shadIdid);`

Vertex Shader

Principe

- C'est un programme qui est exécuté pour chaque sommet du dessin
- Chaque sommet est traité de manière indépendante (pas de notion de polygone)
- On a accès aux propriétés du sommet (position dans le repère de la vue, couleur, coordonnées de texture)
- On fait des calculs serviront lors de l'étape d'interpolation pour fournir des informations aux fragments
- Il faut au minimum fournir:
  - les coordonnées dans le repère de projection du sommet.
  - la couleur du sommet

Vertex Shader

Exemple simple

```
void main(){
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  gl_FrontColor = gl_Color;
}
```

Fragment shader

Principe

- Il récupère les informations interpolées des sommets lors de la phase de tessellation
- Il doit calculer la couleur du fragment `gl_FragColor`

Exemple simple

```
void main(){
  gl_FragColor = gl_Color;
}
```

- `gl_Color` est la couleur interpolée à partir de la couleur des sommets formant la face à laquelle appartient le fragment

Communiquer des informations au vertex shader

Les variables `uniform`

- Ce sont des variables calculées sur le CPU et transmises à TOUS les sommets de la géométrie.
- Ces variables ont la même valeur pour chaque sommet

Variables `uniform` prédéfinies

- `uniform mat4 gl_ModelViewMatrix`
- `uniform mat4 gl_ProjectionMatrix`
- `uniform mat4 gl_ModelViewProjectionMatrix`

Communiquer des informations au vertex shader: `uniform`

Définir ses propres `uniform`

- Dans le code source du shader: déclaration
- Dans le code source du programme: transmission CPU $\rightsquigarrow$ GPU

Code source shader

```
uniform int numFrame;
void main(){
  gl_Position=gl_ModelViewProjectionMatrix * gl_Vertex;
  gl_FrontColor=gl_Color+
                vec4((1+sin(numFrame/100))/2,0.0,0.0,0.0);
}
```

Code source du programme

```
  GLint loc3=glGetUniformLocation(shadPrg,"numFrame");
  glUniform1i(loc3, nbFTot);
```

Communiquer des informations au vertex shader: `attribute`

Les variables `attribute`

- Ce sont des propriétés associées à chaque sommets
- La valeur est a priori différente pour chaque sommets

`attribute` prédéfinis

- `attribute vec4 gl_Vertex;` les coordonnées dans le repère de la caméra
- `attribute vec4 gl_Color;` la couleur du sommet
- `attribute vec3 gl_Normal;` la normale du sommet

Définir ses propres attribute

Code source shader

```
attribute vec3 centres;
void main(){
  vec4 res=gl_Vertex;
  res.xyz+=centres;
  gl_Position=gl_ModelViewProjectionMatrix * res;
  gl_FrontColor=gl_Color;
}
```

Code source du programme

```
  GLint loc2=glGetAttribLocation(shadProg,"centres");
  glEnableVertexAttribArray(loc2);
  glBindBuffer(GL_ARRAY_BUFFER,bufIdent[CENTRES]);
  glVertexAttribPointer (loc2, 3, GL_FLOAT,GL_FALSE, 3*
      sizeof(GL_FLOAT), 0);
```

Communiquer des informations au fragment shader

Les variables `uniform`

- Même principe que pour le vertex shader
- Déclaration dans le source du shader
- Transfert au niveau de programme CPU

Les variables `varying`

- Ce sont des variables calculées par le vertex shader
- Puis interpolées automatiquement lors de la tessellation pour fournir une valeur à chaque fragment

`varying` prédéfinies

- `gl_Color` la couleur interpolée issue des sommets

Communication entre vertex et fragment shader : varying

Vertex shader

```
varying vec3 pos;
void main(){
  gl_Position=gl_ModelViewProjectionMatrix *gl_Vertex;
  pos=gl_Vertex;
  gl_FrontColor=gl_Color;
}
```

Fragment shader

```
varying vec3 pos;
void main(){
  gl_FragColor.rgb=normalize(pos);
  gl_FragColor.a=0.;
}
```